Compiling Communication-Minimizing Query Plans

Eric Love

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-181
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-181.html

December 20, 2019

Acknowledgement

The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000849. Research was partially funded by ADEPT Lab industrial sponsors and affiliates Intel, Apple, Futurewei, Google, and Seagate. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Compiling Communication-Minimizing Query Plans

by

Eric J Love

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Krste Asanović, Chair
Assistant Professor Jonathan Ragan-Kelley
Assistant Professor Zachary Pardos

Fall 2019

The dissertation of Eric J Love, titled Compiling Communication-Minimizing Query Plans, is approved:

Chair ______________________________ Date ____________

______________________________ Date ____________

______________________________ Date ____________

University of California, Berkeley

Compiling Communication-Minimizing Query Plans

Abstract

Compiling Communication-Minimizing Query Plans

by

Eric J Love

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Krste Asanović, Chair

Because of the low arithmetic intensity of relational database operators, the performance of in-memory column stores ought to be bound by main-memory bandwidth, and in practice, highly-optimized operator implementations already achieve close to their peak theoretical performance. By itself, this would imply that hardware acceleration for analytics would be of limited utility, but I show that the emergence of full-query compilation presents new opportunities to reduce memory traffic and trade computation for communication, meaning that database-oriented processors may yet be worth designing.

Moreover, the communication costs of queries on a given processor and memory hierarchy are determined by factors below the level of abstraction expressed in traditional query plans, such as how operators are (or are not) fused together, how execution is parallelized and cache-blocked, and how intermediate results are arranged in memory. I present a Scala-embedded programming language called Ressort that exposes these machine-level aspects of query compilation, and which emits parallel C++/OpenMP code as its target to express a greater range of algorithmic variants for each query than would be easy to study by hand.

Contents

Contents **i**

1 Introduction & Background **1**
1.1 History & Related Work . . . 2
1.2 Databases & Database Machines of Tomorrow . . . 6
1.3 Methodology . . . 10
1.4 Contributions of this Thesis . . . 11

2 Machine-Level Query Optimization **12**
2.1 Introduction . . . 12
2.2 Dimensions of Machine-Level Planing . . . 14
2.3 Plan Derivation by Transformation . . . 15
2.4 Case Study: TPC-H Query 19 . . . 28

3 HiRes: A Language for MLP **33**
3.1 HiRes Operators & Syntax . . . 34
3.2 HiRes Type System . . . 47
3.3 `MetaOps`: HiRes "Macros" . . . 58
3.4 HiRes Case Studies . . . 59

4 Operator Fusion: Constraints & Algorithms **62**
4.1 Introduction . . . 62
4.2 The Shared Clique Formation Algorithm . . . 63
4.3 Inner and Outer Loop Fusion Particulars . . . 75

5 Ressort Compiler Architecture & Implementation **81**
5.1 HiRes Compilation Pipeline . . . 82
5.2 LoRes Compilation Pipeline . . . 91

6 Evaluation **103**
6.1 Experimental Platform . . . 103
6.2 "TPC-H"-Like Queries . . . 104

7 Communication-Aware Query Cost Models 109

7.1 Original Model of Manegold, Boncz, and Pirk . . . 109

7.2 Extensions for Communication Bounds . . . 112

7.3 A Manegold-Based Roofline Model . . . 121

7.4 Summary & Limitations . . . 122

7.5 Case Study: TPC-H Query 19 . . . 123

7.6 Proof of NP-Hardness of Single-Phase Allocation . . . 127

7.7 2x Error Bound on Convex Hull Allocation . . . 128

8 Conclusions & Future Work 130

8.1 Summary . . . 130

8.2 Principal Findings & Recommendations . . . 131

8.3 Limitations & Future Work . . . 131

8.4 A Methodological Plea . . . 132

Acknowledgments

The greatest reward by far of graduate school has been getting to know the friends and colleagues I met along the way, and this thesis would not have been possible without their support, encouragement, patience, and advice. I will remain forever grateful to more individuals than could be named here, but I would like to give special thanks to a few:

- I want to expresss my gratitude to my advisor, Krste Asanović, who generously supported my forays into research outside the more traditional realms of computer architecture and hardware design, as well as my other committee members Jonathan Ragan-Kelley and Zachary Pardos, who were kind enough to review this dissertation.

- While a post-doc at UC Berkeley, Lisa Wu Wills was a mentor to me and advocate of this work. I am happy to see her advise many other graduate students now as faculty at Duke University.

- I am lucky to have met an exceptional generation of senior students when I arrived in Berkeley. Scott Beamer and Chris Celio were two of the first people to welcome me, and our friendship has only grown through three successive labs, numerous RTL refactoring sessions, and more than a healthy number of session ales. Practically everything I know about micro-architecture I learned from discussions with my former and future colleague Andrew Waterman–more likely than not in our local watering hole, Beta Lounge, whose own contribution to this thesis is incalculable.

- Upon entering the PhD program together, Martin Maas and I shared a desk, a research project (PHANTOM), and many fond memories. My subsequent desk-mate, Orianna DeMasi, is an inspirational human and inexhaustible source of encouragement. I am grateful for my continued friendship with her and our friends Albert Magyar and Nathan Pemberton, without whom this thesis would not have been possible.

- The UCB-BAR research group has been an overwhelmingly fun workplace, and I am fortunate to have had such outstanding colleagues there, among them Adam Izraelevitz, Albert Ou, Alon Amid, Colin Schmidt, David Biancolin, David Bruns-Smith, Henry Cook, Howie Mao, Jack Koenig, Palmer Dabbelt, Sagar Karandikar, Sarah Bird, and Yunsup Lee. While it is sad to say goodbye, am excited to continue working alongside some of them at SiFive.

- Of course, some of the most important people in our lives may sit, not next to us, but thousands of miles away, and I do not know how I would navigate life without the daily virtual presence of Rose Nissen, my best friend since high school and an indefatigable force. And though the end of college has put a whole continent between us, Dylan Morris and Kavita Mistry have been consistent companions throughout the tribulations of grad school.

- I would not be here without a lifetime of encouragement from my parents, Jack and Susan, to whom I owe everything, and who, in reading this, will be elated to see that I have finished. Neither would I have completed a computer science PhD without the mentorship of Yiorgos Makris, who encouraged me as an undergraduate to pursue research and gave me my first opportunities.

- Lastly, I would like to thank the sponsors of the ADEPT Lab, whose generous funding made this work possible. The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000849. Research was partially funded by ADEPT Lab industrial sponsors and affiliates Intel, Apple, Futurewei, Google, and Seagate. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Chapter 1

Introduction & Background

This thesis takes a step towards demarcating the performance limits to the performance of database analytics queries on the hardware of today and of the next decade. Such workloads, whether known as *decision support* queries, *data warehouse* queries, or *online analytics processing (OLAP)*, do largely what is summarized by the SQL in Figure 1.1, the 6th query from the TPC-H [70] benchmark suite (Q06). They scan large arrays of data, compute various quantities derived from them, filter them based on those results, and tabulate some final answer. Other queries may be more complicated, and incorporate relational joins, but this basic pattern constitutes the bulk of the "work" done in analytics processing.

```sql
select
        sum(l_extendedprice * l_discount) as revenue
from
        lineitem
where
        l_shipdate >= date '1994-01-01'
        and l_shipdate < date '1995-01-01'
        and l_discount between 0.06 - 0.01 and 0.06 + 0.01
        and l_quantity < 24
```

Listing 1.1: TPC-H Query 06 in SQL Form

While they may seem computationally simple, significant complexity lurks wherever their peak performance is sought. And the throughput of queries similar to Q06 has been important enough to motivate the design of custom hardware for their acceleration since at least a century and a half ago. The principal concern of this thesis is how to approach the design of database hardware for the next decade. However, we do not advance any particular architecture or accelerator blueprint, and instead argue that the first step towards any such effort should be to *consider the widest possible space of semantically equivalent algorithms (query plans) that might be employed to implement the range of queries likely to be processed by the new hypothetical machine, and only then to propose hardware with the "best" of these algorithmic mappings in mind*, rather than optimizing (micro-)architectures for whatever implementation strategy happened to have superior performance on platforms that exist right now. The notion of optimality in this case turns out to be minimal data movement, as Setion 1.2 later argues.

Consequently, we propose a language and compile framework that widens the space of hardware-conscious plan choices that query processors can consider, and attempt to predict how much future hardware could improve its results.

1.1 History & Related Work

Before contemplating the contemporary obstacles to OLAP optimization, though, it is instructive to review some characteristic examples from the last hundred years of attempts to execute analytics queries at ever-greater speeds. At each stage, what changes is the physical constraints of the underlying hardware.

A Century and a Half of Hardware-Driven Analytics

In the 19th Century

Already in 1888, Herman Hollerith's tabulation machine [69] was enlisted to analyze the results of the upcoming American census of 1890. Though it lacked any Turing-complete processor, it accelerated aggregation "queries" by advancing mechanical dials one step at a time in response to electrical signals formed where metal springs passed through holes punched in a paper card to indicate the presence or absence of demographic characteristics. It also facilitated group-by aggregations using a "sorting drawer" into particular slots of which operators were instructed to insert cards based on some combination of their attributes. Moreover, this specific machine was selected for use in the Census because it excelled in an already-standardized query benchmark contest, comprising the tabulation of prior census results for some districts of St. Louis, M.O. Thus, even without the benefit of general-purpose computers, a purely application-specific accelerator approach to query processing prefigured today's landscape of database hardware proposals in both its approach and its metrics.

In the 20th Century

Nearly one century after Hollerith's pioneering efforts, hardware support for analytics processing was deemed of sufficient importance to merit a special issue of the IEEE Transactions on Computers [30] in 1979. In an era of mainframes, tape drives, and relatively expensive magnetic disks, "the database machine (DBM) [was] the result of an atchitectural approach which [...] distributes processing power closer to the devices on which data are stored." Four decades ago, then, the idea emerged of processing near data as a way to eliminate the CPU as a central bottleneck to the processing of data at the aggregate bandwidths supplied by storage devices; today, proposals for intelligent SSD controllers echo this idea.

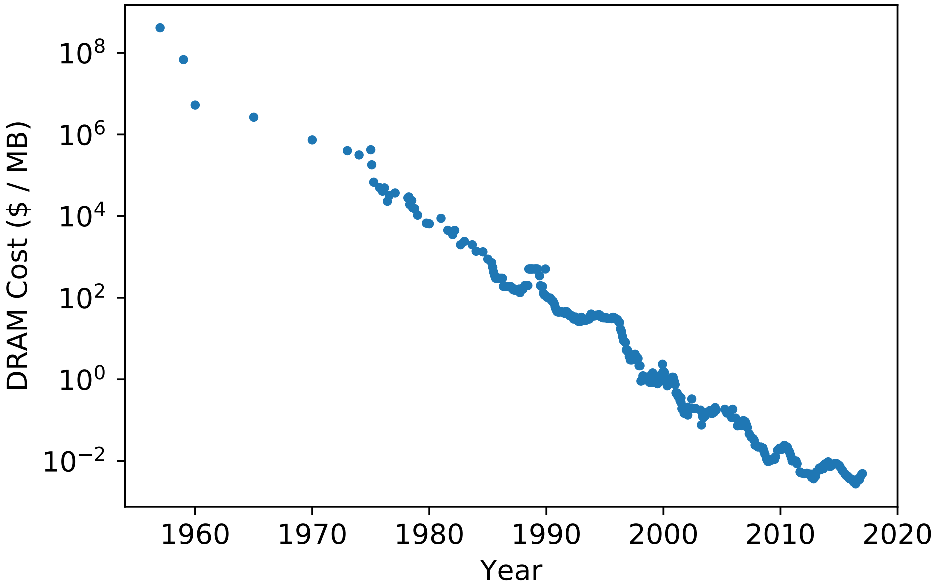


Figure 1.1: The Cost of DRAM Over Time (Data from McCallum [21])

A decade later, the Gamma Database Machine project by DeWitt et al. at the University of Wisconsin witnessed the dawn of the main memory era, in which multiple megabytes of DRAM were inexpensive enough to provision for a single compute node, as Figure 1.1 shows. In their proposal, dozens of Intel processors interconnected with a custom network could perfom parallel joins by partitioning hash tables into chunks small enough to fit (sometimes) in the 8MB of DRAM attached to each processr. Their software also compiled some selection predicates to executable code, foreshadowing the age of query compilation in which SQL interpretation overhead was an obstacle.

In the 21st Century

By the turn of this century, the cost of DRAM had dropped to mere dollars per megabyte, making gigabytes available per processor, and fitting entire tables into main memory. The resulting reduction in disk seeks and increase in scan bandwidth quickly made CPU power a key performance barrier in a growing number of analytics workloads. This led to the emergence of *column stores*, which packed all rows of a single attribute together in continuous extents of memory. This enabled the application of a relational operator to whole *columns* of records at a time, whereas under previous execution models, records would have been processed individually, and any control or interpretive overhead for each operator was incurred repeatedly. For example, in the so-called "Volcano" model (after the project of the same name by Graefe [16]), each operator instantiated in a query plan is an object supplying a `next()` method that downstream operators can call to obtain one tuple at a time, and which itself would call its own inputs' `next()` methods, incurring whatever function call overhead this entailed many times per record. By contrast, C-Store [66] (later Vertica [28]) and MonetDB [7] (later Vectorwise [76]) were early prominent examples of column stores that upended this paradigm.

The transition from record-at-a-time to operator-at-a-time query processing also facilitated the exploitation of instruction- and data-level parallelism inside the processor, becuase each operator now consisted of a single tight loop over records that could be unrolled either dynamically by superscalar pipelines or statically by compilers, and in some cases this even meant that vectorized instructions could then be emitted. This, in turn, led to a proliferation of papers explaining how best to hand-optimize common relational operators for emerging multi-core [26, 5, 3] and multi-socket [29, 59] systems, as well as SIMD instruction set extensions [57, 71, 51, 48].

Even column-at-a-time execution suffers from the requirement that each operator's result be written out fully to memory, before being read back again by the next operator that consumes it. One way around this problem, implemented by Vectorwise, was to switch from column-at-a-time processing to *block-at-a-time* processing, where each block would be small enough to fit in cache. In this way, dependent operators would pass data to each other through cache, at least, rather than memory, while still amortizing control overhead across each block.

Eventually this too proved inadequate, as the highest-performance database architectures sought to bypass the memory system entirely and pass data between operators through registers. This required query *compilation*, and several projects emerged to turn SQL into executables, most notably HyPer [25] by Kemper and Neumann, which used the LLVM [31] compiler backend to generate machine code. HyPer was eventually commercialized as Tableau [67], and since then several other academic query compilers have appeared, focusing either on how to structure such compilers [60, 68], or how to tune code-generation parameters [9].

At the same time, need arose for standard benchmarks for analytics. The Transaction Processing Council, which had for nearly a decade supplied the industry standard benchmark for transaction processing performance, devised the "H" suite of about two dozen non-transactional queries [70]–such as that shown in Listing 1.1. This has animated much of the research into both hardware- and software-based analytics support ever since, and this dissertation makes use of it as well, though without following all of its requirements (see Section 1.3 below).

Database Machines Today

More recently, growing demand for OLAP performance and even lower costs per byte of DRAM have inspired processor manufacturers to design analytics-tailored hardware. Intel collaborated with SAP to offer database-oriented servers capable of addressing up to 12TB of memory from eight sockets [20], while Oracle's SPARC M7 [43] supports 16TB, and includes instruction set extensions designed specifically to support analytics by accelerating scans, filtering, joins, and compression.

Aside from the instruction-level acceleration offered by Oracle, these offerings mostly resemble traditional, general-purpose server platforms with board-level support for larger memory capacities (i.e. custom buffer chips to access more DIMMs and more threads per socket). However, as Moore's Law [41] appears to reach its end, and the number of transistors that can be packed on a single chip at optimal cost no longer doubles every eighteen months, more application-specific kinds of acceleration will have to replace raw core counts as the main driver of higher query throughput.

Accordingly, academics have investigated alternative architectures and micro-archtectures. The most general-purpose of these add, like the SPARC M7, assistance for various aspects of query processing. Hayes et al. proposed extensions to vector instruction sets to ease the cache pressure that results from vectorizing radix partitioning and joins [17, 18]. In "Meet the Walkers" [27], Kocberber et al. offloaded the serial indirect loads (pointer chasing) of hash-table builds and probes to dedicated functional units that live outside of the processor's pipeline, allowing many more memory operations in flight than an out-of-order core could sustain. At the more application-specific end, Kara et al. proposed an FPGA-based partitioning accelerator [23], while Wu et al. presented a dataflow architecture for accelerating entire queries on FPGAs with networks of operator-specific accelerators [74]. Finally, Oracle's Database Processing Unit (DPU) was published at MICRO'17 [1], and combined ensembles of 160 small, in-order cores per chip with a 76GB/s memory interface to perform hash partitioning and other relational operations at line rate.

The efficacy of these proposals has been judged both by way of the TPC-H benchmark suite, and by others that emerged to address some of its limitations. Leis et al. proposed a Join Order Benchmark (JOB) [33] to increase the number of joins in sample queries up to eight or more, unlike the two or three typical of TPC-H, and further proposed the use of real-world datasets from IMDB instead of the synthetic relations with uniform random values specified by TPC-H, as these would be more reflective of the queries today's users seek to run in practice. In order to facilitate research into database hardware, Shao et al. proposed DBmbench [61], a suite of microbenchmarks that better represent analytics query execution with small datasets than would TPC-H scaled to an equivalent size, which would result in e.g. smaller hash tables and therefore different micro-architectural characteristics. The work that follows neither makes use of the Join Order Benchmark nor DBmbench, but does contemplate new standards for evaluating query performance in the future.

1.2 Databases & Database Machines of Tomorrow

In light of the foregoing decades' worth of query acceleration research, it is reasonable to ask what a database machine of the next decade might look like, and how one might go about designing it. The downward trend of Figure 1.1 suggests that the cost of DRAM will only continue to shrink. Meanwhile, the improving performance of emerging non-volatile memory (NVM) technologies, like Intel's Optane DC Persistent Memory Modules (PMMs) [19], will put terabytes of data within a few hundred nanoseconds of processors, which database researchers are beginning to exploit [4, 53]. What sort of system architecture would best be poised, ten years hence, to make use of this abundance?

Limits to Query Processing Performance

The answer may depend on one further trend in future memory systems: the relatively stable *bandwidths* that electrical interconnects can supply between processors and off-chip memory, which already are scaling more slowly than transistor densities. Memory capacities continue to rise through the terabyte range, growing numbers of DIMMs to support them should supply higher aggregate bandwidths even if single DRAM bank cycle times stay fixed. However, individual chips are limited in the amount of data they can transmit per unit time via the pins affixed to their packages, whose size does not scale downwards with transistors, and whose cost remains high relative to other parts of the chip. This limit has remained within 100-200GB/s per socket over the past few years for the highest-end server parts (such as the Intel and Oracle DBMs in Section 1.1), and in general, over the past decade, the ratio of arithmetic capacity to bandwidth has been increasing at a rate of about 20% per year [39], meaning that a so-called *bandwidth wall* likely lies ahead.

Arithmetic Intensity of Analytics Queries

TPC-H Q06 query hints at why this bandwidth wall represents a crucial impediment to analytics performance. Its simplicity makes it very easy to calculate how much data a (naïve) plan for that query must move between processor and memory: assuming the four attributes accessed are each encoded as 32-bit values (two single-precision floating point numbers and two integer-valued dates), then 16 bytes must be scanned per record, for which 7 operations (5 comparisons, one multiply and one add) must be performed, yielding an *arithmetic intensity* of 0.44 ops/byte. In Figure 1.2, a *roofline model* [72] plots the performance limits of different machines as a function of a workload's arithmetic intensity. Q06 lies *far* to the left, bandwidth-bound side, regardless of whether one considers the roofline for an Intel Xeon E5-2667 (Broadwell) server part or Google's custom Tensor Processing Unit (TPU) [22] (chosen to show two ends of the machine roofline spectrum). [1]

The latter, whose crossover point at which workloads can start to become compute-bound lies beyond 1000 (floating point) ops/byte, is shown to emphasize one point: that hardware acceleration tends to succeed *only when arithmetic intensities are orders of magnitude higher than that of Q06.* Moreover, since empirical measurements of this particular query's performance later on (Chapter 6) show that existing machines already achieve close to 50% of the roofline's indicated peak, no amount of custom acceleration hardware could accelerate that query plan more than 2x unless more bandwidth can be supplied.

[1] The Oracle DPU [1] would be a more natural comparison here, but we could ascertain from their publication how much arithmetic bandwidth each DPU die supplies.

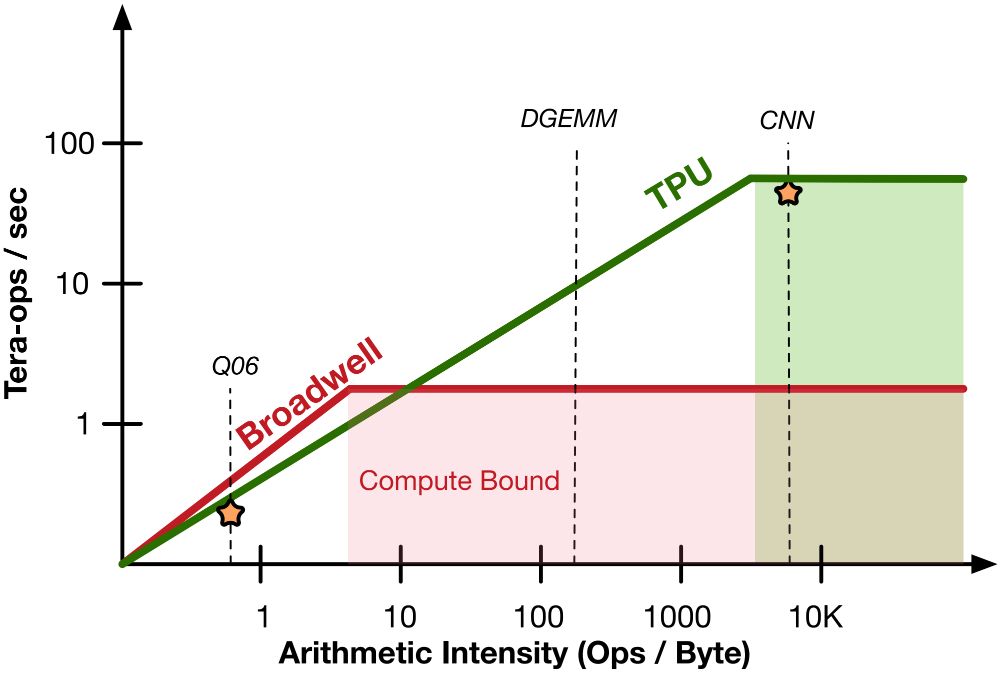


Figure 1.2: Approximate Roofline Model of Intel Broadwell and Google TPU

Towards the Bandwidth Wall

Q06 may have been a particularly egregious example, since it consisted of an entirely scan-based query, but a similar analysis can be extended to more complex analytics operations as well. Joins, for example, ought to entail more significant data re-use. Using a more elaborate analysis than the above (see Chapter 7), we have estimated the communication costs–and therefore peak throughput–of the high-performance hash-join implementation presented by Schuh et al. [59]. Relative to the specifications of the machine reported in their evaluation, their measurements already appear to be approaching 40% of the bandwidth-defined peak, for at least some problem sizes. Similar analysis for other operators–especially using the AVX256 vectorization results of Polychroniou et al. [48]–gives even higher fractions of the theoretical machine peak. From this we might extrapolate that the next few years of research, combined with the increasing prevalence of "SQL-in-Silicon" [43] instruction extensions, will propel relational operator implementations in high-performance query execution engines ever closer to their communication-determined limits.

Implications of the Bandwidth Wall

These trends suggest that designers of tomorrow's DBMs should focus less on hardware extensions to accelerate individual relational operators and more on customizing the compute and memory systems to help queries *move less data*; secondly, they should concentrate on the possibilities for reducing memory traffic that arise when *full query compilation is employed*, and more inter-operator optimization is possible.

Hardware-Assisted Communication Reduction

Within the former category, designers may consider at the coarsest level of granularity whether a database-focused processor might for example benefit more than other workloads from devoting a larger fraction of fixed silicon budgets to cache rather than pipelines, or whether analytics workloads can benefit from technologies like High-Bandwidth Memory (HBM) [36, 47], which places multiple gigabytes of DRAM into the same package as the processor, enabling much higher access bandwidths[2]. The utility of either approach depends on finding temporal locality (data re-use) in query execution, which itself depends critically on choices made during query compilation.

Specialized hardware can also be employed to reduce the amount of data that must be brought into the processor in the first place, for instance by performing filtering operations in memory or SSD controllers [13], as some have explored. Or, designers can consider whether DBMs might best be constructed with a larger number of smaller sockets, supplying greater aggregate bandwidth albeit with a higher likelihood of NUMA effects. We forego such considerations in this study, on the grounds that, eventually, what happens within a single package still matters.

Some queries like Q06 can really only be sped up if accesses to some fields can be avoided, or their size can be reduced. To avoid unnecessary accesses, the column imprints technique of Sidirourgos and Kersten [63] supplies a small secondary index structure to avoid accessing cache lines if their contents can be determined not to contain relevant values. In complex predicates, accesses to unneeded attributes can be pruned in the presence of short-circuit boolean ANDs, as 2 describes, though this can result in steep branch-misprediction penalties on general-purpose processors. Meanwhile, compression techniques that trade additional arithmetic for less data movement are another way to reduce traffic, and they can be made cost-free with hardware support. For queries that use string-valued attributes, dictionary compression can drastically reduce data movement when the domain of an attribute is small. This can be combined with *bit packing*, which places multiple elements of non-byte-sized (e.g. 6- or 11-bit) values contiguously into the same word, requiring unpacking upon access, which Polychroniou and Ross [50] showed how to do in isolation at line rate on Intel architectures using AVX SIMD instructions, but which in the context of more complex compiled queries could likely benefit from hardware support. BitWeaving [34] solves a similar problem with an even more computationally expensive mechanism.

Clearly, compression and avoidance techniques should be assumed in the design of a future DBM, but the more interesting implication of the approaching bandwidth wall is that database hardware should *seek to accelerate algorithms and query plans that minimize*

[2] Though not that much higher: the HBM2E parts announced by Samsung [56] in 2019 offer 410GB/s, which does not represent an order of magnitude difference with regard to the highest-end off-chip memory products

data movement, rather than those that simply happen to be the fastest already on today's platforms. This might entail tolerating *random* accesses better than general-purpose out-of-order cores, as would be necessary to make full use of technologies like HBM [47]. This consequence of the bandwidth constraint is the focus of the rest of this thesis.

Communication-Minimizing Query Plans

While the arithmetic intensity of individual relational operators is not high relative to that of other workloads, the trend towards dynamic compilation offers new opportunities to treat *entire query plans* as single algorithmic units to be optimized and, conversely, for which to optimize hardware. Not only does compilation make it possible to reduce communication by avoiding the materialization of intermediate results, but it means that the memory-aware optimizations employed by Schuh et al. [59] (and others) to implement individual operators can be codified into re-usable *code generators* that can extend their effects across fused sub-queries while also tailoring them as needed to any combination of input datatypes and formats. Since the consequences of such choices intimately affect a query's interaction with the hardware on which it executes, we call them *machine-level plan* decisions, and discuss them at length in Chapter 2. And since they can significantly alter the query's data movement requirements in particular, we think that the large plan space they engender should be studied intensively by the architects of any future DBM. Furthermore, *those plans, or plan choice strategies, that result in the least communication should be the ones DBMs should be designed to accelerate.*

Secondly, it is possible to analyze these communication requirements for a given memory hierarchy and degree of parallelism without physically measuring the number of bytes transferred on a physical machine. The cache-miss cost model of Manegold, Boncz, and Kersten [38] can be leveraged, in modified form, to estimate the number of cache lines that a query plan–when specified at a sufficiently low level of abstraction–should need to transfer. If the widest possible space of (machine-level) plans for a query is analyzed in this way, then the least communicative of these ought to be considered as an *upper bound* on that query's performance for any system with the same bandwidth and cache capacity. If that plan does not turn out to be the empirically highest-performing on today's systems, then the task of a DBM designer is to remove whatever computational bottleneck prevented it from executing at line rate. At the same time, the results of such study should indicate the answers to questions about the marginal benefit different classes of query can derive from different allocations of cache and compute resources, making it possible to co-design the hardware with query plans.

More generally, it can be hoped that future advances in both hardware and software approaches to query acceleration be judged relative to the bounds set by the bandwidth wall, and made knowable by communication cost models. As expectations about query benchmarks continue to evolve, so too should the methods of interpreting those benchmarks' results, and the ensuing chapters take initial steps towards that goal.

1.3 Methodology

A few necessary caveats limit the scope of this study:

- **No DB frontend**: We cannot compile all the way from SQL directly, as our current prototype lacks a SQL parser, algebrizer, and logical optimizer.

- **Limited benchmark set**: We do not present results for any complete benchmark suite–neither TPC-H nor TPC-DS nor others–but rely instead on a limited subset of queries for which we could produce plan generators.

- **Non-conformance with TPC-H**: To the extent that we make use of "TPC-H" queries, we do not adhere to all of the specification requirements. In particular, all string-valued attributes are assumed to be *dictionary-encoded* to the smallest byte-aligned size that can contain them, and consider only queries for which this is possible.

- **Oracular cardinality estimation**: Any discussion of "optimal" plan choices assumes that query optimizers can predict the selectivity of any predicate, which in practice is not possible.

- **Throughput, not latency**: Compilation time (both in the Ressort compiler and in C++ compilation) is excluded from any benchmarking on real systems, as the primary focus is on maximizing the rate of processing large datasets

- **Lock-free algorithms**: While many parallelization strategies for the algorithms under discussion have been devised that rely on either locks or atomic operations, these are deferred to future evaluation in favor of their lock-free alternatives

- **Raw performance**: Even if the bandwidth wall stalled query throughput entirely, custom architectures for analytics could easily increase performance per Watt of the processor, but we do not consider power or energy normalization here. (It should be noted, though, that even a zero-Watt processor could not achieve orders-of-magnitude energy efficiency increases for workloads that saturate memory bandwidth, as the latter subsystem's contribution to overall power footprint is substantial.)

1.4 Contributions of this Thesis

In light of the aims outlined above, the main contributions of this thesis are: (1.) the introduction of the concept of *machine-level plans* (Chapter 2), which unify all communication-determining aspects of fully-compiled queries into a single abstraction, (2.) the design of a statically-typed domain-specific language, *HiRes* (Chapter 3), for representing machine-level plans, and (3.) a compiler, *Ressort* (Chapter 5), that refines it to C++ code with parallel OpenMP [11] annotations that can be compiled and executed with any existing toolchain. Additionally, Chapter 7 presents a set of refinements to the communication cost model of

Manegold, Boncz, and Kersten that expand its predictive capability to future memory systems.

Chapter 2

Machine-Level Query Optimization

2.1 Introduction

Fully-query compilation, in which SQL programs produce executable binaries, is the state of the art for high-performance, in-memory query processing engines, as it removes any per-record SQL interpretation overhead, allows for the exploitation of instruction- and data-level parallelism within queries, and permits inter-operator optimizations that would be impossible in any other execution model. Many recent papers have presented advances in the design and architecture of query compilers [25, 9, 60, 68] and intermediate representation formats [46, 44] for query plans, making them an active area of current research. At the same time, the database community has investigated in great depth how to optimize hash joins [26, 3, 59] and other relational operators [49, 51] for the lowest-level of processor-specific parameters such as cache geometries, branch predictors, and vector instruction set extensions. While query compilation frameworks generally acknowledge the importance of these last-level mapping considerations, and set their parameters at a coarse granularity, not all the proposals of these works are fully expressible using the lowest level of query representation such compilers expose. This thesis builds on the rich legacy of research on hardware-tuning relational operators to design a new, more expressive query compiler back-end.

More specifically, it introduces the concept of a *machine-level plan (MLP)* representation, which makes explicit all the memory-resident data structures needed for query execution, along with their layout (e.g. row- or column-ordering for each subset of attributes in a relation), the manner in which they are accessed, and how execution is parallelized. The choices presented by MLP lie below the traditional level of query optimization, where relational algebra may be rewritten to, e.g., change the order of joins or the application of `where` clauses. Once those choices are fixed by a traditional query optimizer, MLP plans answer questions such as "which attributes in a scan can be skipped based on already-known predicates?", and "when are the results of selection compacted into contiguous extents?'" and "which attributes used after a join are packed into a custom hash-table?" and "where if at all in a query should partitioning be inserted to induce parallelism or locality?" Even once these questions are settled, MLP representation exposes yet another plan tradeoff, namely: which buffers must be physically materialized in memory, and which may exist only virtually, in CPU registers? This chapter shows how the both the traditional Volcano (record pull) [16] and data-centric (record push) processing models enforce one particular set of answers to this question, while query semantics permit others that may be more optimal.

The rest of this chapter explains the idea of machine-level plans more thoroughly, describing first the dimensions along which machine-level plans can be varied, how plans can be generated by transformation from a baseline, what the relative costs and benefits of each choice are, and, finally, how such choices can combine across a full plan. Put another way, this chapter codifies the toolbox of optimizations appearing in hand-tuned operator implementations that others have described, and makes them available for full queries and for re-targeting to inputs with arbitrary datatypes and formats. *We do not claim to have invented them, but contribute instead a synthesis of these tuning methods, into a generalizable framework.* The discussion in this chapter serves to motivate the specific machine-level plan representation presented in Chapter 3.

Assumptions

In the presentation that follows, the performance implications of machine-level planing choices are considered relative to two abstract machines. One consists simply of an idealized on-chip storage unit surrounded by idealized compute resources and connected to memory by a channel of fixed and finite bandwidth. It is limited only by the need to move data between on-chip and off-chip memory, and represents a purely bandwidth-bound query processor. Tradeoffs for each plan choice are discussed first relative to this *memory* or *traffic* bound.

The other abstract machine is a server-class, multi-socket, general-purpose multiprocessor representative of the machines typically used for high-performance analytics processing today, and provides context for computational (i.e. not bandwidth-related) bottlenecks that limit the throughput of query plans on hardware that exists today and which was available for evaluation. Tradeoffs for machine-level planing are secondarily discussed relative to platforms with real-world artifacts such as branch predictors and load-store queues, as well as a limited arithmetic bandwidth that requires parallelism to be fully utilized. Units of parallel execution are referred to as "threads" for the sake of convenience, but could represent some other abstraction such as SIMD lanes.

A much broader spectrum of compute devices exists, and many of these have been considered for query processing, from GPUs [52, 57, 75, 8] to FPGAs [73, 74]. We omit explicit discussion of these platforms, as many insights from the compute-bound analysis are more or less translatable. Instead, we turn our attention to the most common query processing machines, and those yet to be built.

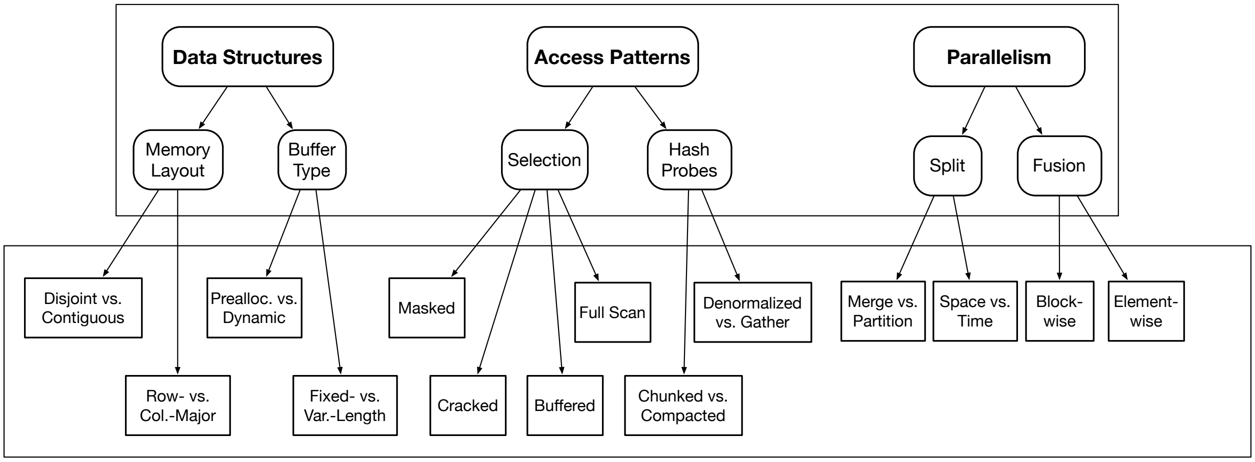


Figure 2.1: Dimensions of Machine-Level Plans

2.2 Dimensions of Machine-Level Planing

The upper box of Figure 2.1 shows the three basic "dimensions" along which machine-level planing decisions can be made: *data structures*, *access patterns*, and *parallelism*. These dimensions are not, strictly-speaking, orthogonal, and choice of one may constrain choices of another, but they are a nonetheless a useful decomposition of the ways in which a query's execution maps onto memory hierarchies and computational resources.

Data structures. In the course of execution, various intermediate results will need to be *materialized* by some operators into memory-resident buffers for use by others. Even if a query's original inputs have a fixed structure of independent, dictionary-compressed, per-attribute columns, these intermediate products do not need to be arranged the same way, but different memory layouts, such as row- and column-major order, or non-contiguous buffers (e.g. those split among NUMA domains), offer different access performance characteristics. Meanwhile, some buffers may be designed to accommodate an initially-unknown number of elements, while others may require counting their maximum storage requirements before use, but allow easier access afterwards.

Access patterns. Some kinds of operations result in different orders of access to data that is already arranged in memory. For example, selection (filtering) operations can obviate the need to read individual rows of buffers used in subsequent steps, but avoiding such reads may result in additional computation, or require buffering indices of valid rows, and then gathering those that make the cut. Structures that are *probed*, like hash-tables, may start off with linked lists of elements that must be traversed, but can be reshaped to flattened forms after insertion is completed, in order to allow contiguous probes.

Parallelism. Individual operators can be parallelized by dividing up their inputs between threads, or partitioning them based on data-dependent key values, and either choice may affect how subsequent operations on their results are parallelized. Two or more operations can be *fused*, interleaving their execution on an element-by-element basis without any intervening materialization, or by interleaving the processing of smaller blocks of the elements processed by each (i.e. the block-at-a-time model implemented by Vectorwise [76] as described in Section 1.1 of the previous chapter. These choices can change communication patterns and cache working-set sizes substantially.

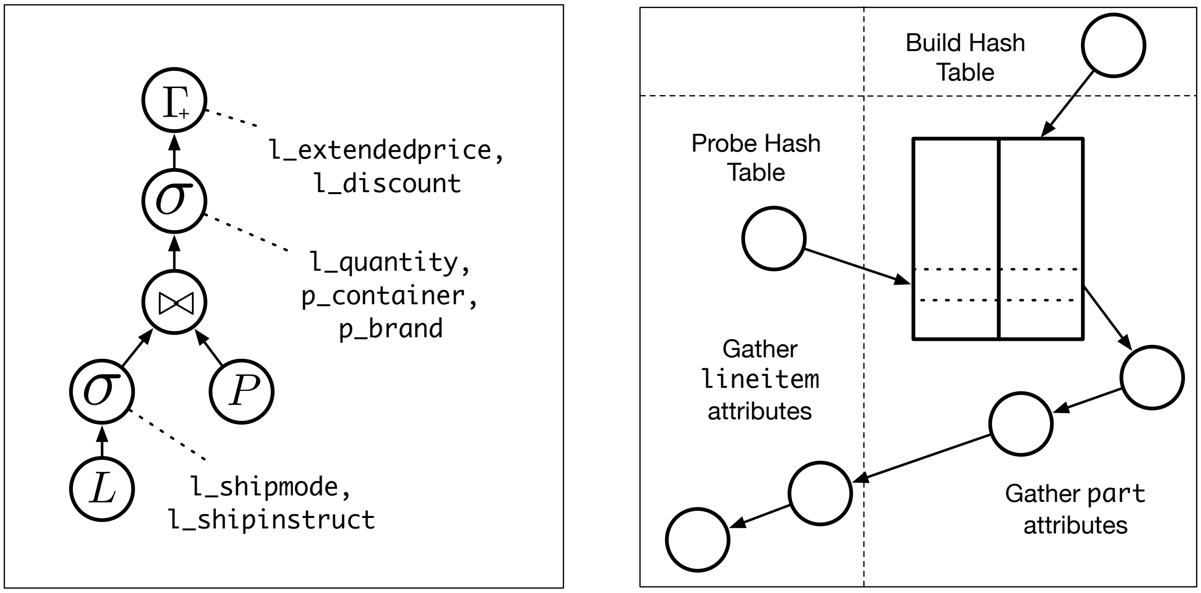
Within each of these categories, the more specific parameters shown in Figure 2.1's lower box can be varied by *transforming* a default, or baseline, plan into more complex ones by a series of procedures that require very different realizations as compilable code. Without presenting a formal representation of query plans at the machine level, the next section reviews these transformational possibilities abstractly, and describes the benefits they may provide in terms of communication and computation efficiency, as well as costs they might entail of the same resources.

2.3 Plan Derivation by Transformation

To illustrate the concrete choices made in each of the three machine-level planning dimensions, consider a query such as TPC-H Query 19, whose SQL is shown in Listing **??**. In classical query planning, a "logical" optimizer would fix the order in which selections occur before or after a join (in this case realizing that each of the `l_shipmode` and `l_shipinstruct` clauses is identical and can be factored out before the join), resulting in the plan of Figure 2.2a. A "physical" planner would choose to instantiate a hash join, for instance, instead of a nested-loops join.

```
select
  sum(l_extendedprice* (1 - l_discount)) as revenue
from
   lineitem,
   part
where
  (p_partkey = l_partkey
    and p_brand = ’Brand#12’and p_container in (’SM CASE’, ’SM BOX’, ’SM PACK’, ’SM
    PKG’)
    and l_quantity >= 1 and l_quantity <= 1 + 10 and p_size between 1 and 5
    and l_shipmode in (’AIR’, ’AIR REG’) and l_shipinstruct = ’DELIVER IN PERSON’)
  or
    (p_partkey = l_partkey and p_brand = ’Brand#23’
    and p_container in (’MED BAG’, ’MED BOX’, ’MED PKG’, ’MED PACK’)
    and l_quantity >= 10 and l_quantity <= 10 + 10 and p_size between 1 and 10
    and l_shipmode in (’AIR’, ’AIR REG’) and l_shipinstruct = ’DELIVER IN PERSON’)
  or
    (p_partkey = l_partkey and p_brand = ’Brand#34’
    and p_container in (’LG CASE’, ’LG BOX’, ’LG PACK’, ’LG PKG’
    and l_quantity >= 20 and l_quantity <= 20 + 10 and p_size between 1 and 15
    and l_shipmode in (’AIR’, ’AIR REG’) and l_shipinstruct = ’DELIVER IN PERSON’);
```

Listing 2.1: TPC-H Query 19 in SQL Form



(a) Traditional Query Plan (Semi-Physical) for TPC-H Q19: Scan of `lineitem`, followed by selection on `l_shipmode` and `l_shipinstruct`, hash join on `l_partkey`, followed by further selection and aggregation (average).

(b) Base Q19 Execution Diagram. Circles represent column scans, and arrows represent the "flow" of a tuple, while boxes represent a hash-table or materialized partition.

Figure 2.2: Two views of a default plan for TPC-H Query 19

Then, a "machine-level" planner would further refine this into something like Figure 2.2b, which depicts a query plan's execution in terms of the flow of a record through space and time, as well as the layout of relevant data structures, rather than as a DAG of relational operators, in order to reveal the plan's machine-level characteristics.

Here, circles represent accesses to one or more attribute columns (scans and gathers) and the computation of any results from them, while arrows represent actions that each record may conditionally *cause*, such a subsequent scan, a hash-table probe, or insertion. Square boxes denote hash-tables, while dotted vertical lines very coarsely separate the two relations (the *build relation* `lineitem` and *probe relation* `part`). Horizontal dotted lines divide execution into distinct stages; in Figure 2.2, the "hash-table build" stage must complete before the records of `lineitem` are joined against its result. Thus, these time divisions correspond–roughly–to separate loops or loop nests in any code that would implement them.

In Figure 2.2b, two relations are joined via a hash-table built from a single key attribute, preceded by and followed by selection based on others, which are *gathered* (via position vectors) from their original columns when needed. The join style is a reasonable default, as many database engines–especially non-compiled ones–will have a ready-made join operator that expects a (hash, row-ID) tuple table format.

This example is fairly representative of many other queries, which differ merely by the addition of further joins, selections, and aggregations. In each of the sections that follow, one machine-level plan *transformation* offers a choice that affects the execution of one of this plan's operators; later, Section 2.4 shows how these choices interact across the entirety of a machine-level plan, and how they can apply recursively.

#1: Split-Merge Parallelization

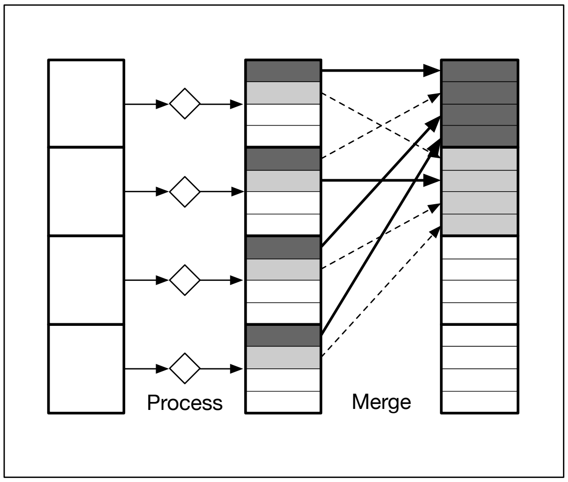
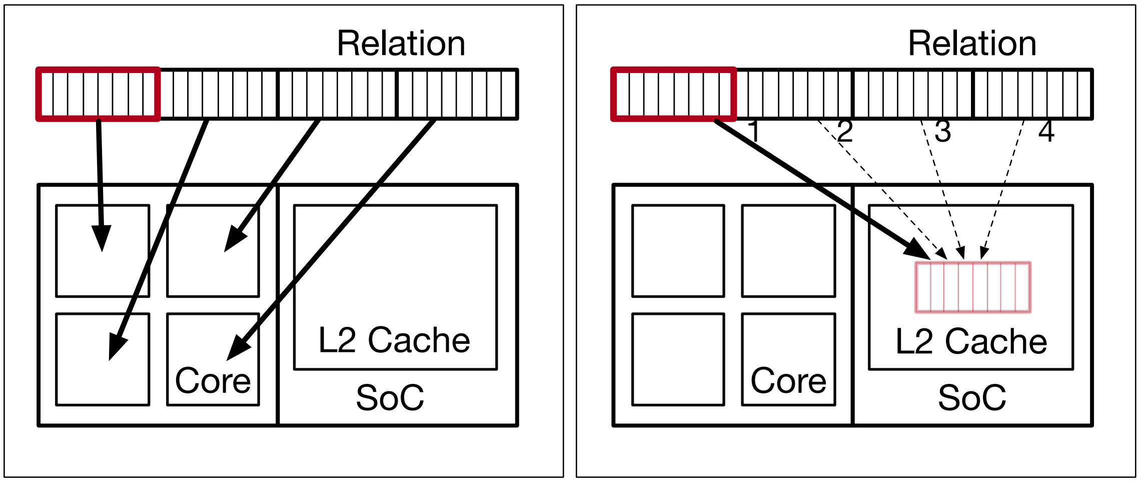


Figure 2.3: Split-Merge Parallelization

The simplest way to parallelize a query plan is to *split* a relational operator's input records into equal-sized subsets, and apply the operator to those subsets separately. The kind of independence imposed on the resulting subsets can either be *spatial* (Figure 2.4a), or *temporal* (Figure 2.4b). Spatial parallelism applies when each unit of input is assigned to a different processing element, such as a thread, processor, socket, or SIMD lane. Temporal parallelism applies when the units of input are still processed sequentially in time, but the scope of inputs considered at once is limited in order to reduce contention on some resource such as cache. The cache blocking of matrix multiply algorithms is an example from another domain; in relational query optimization, an example would be building independent hash-tables for subsets of the total input under the assumption that these will fit better in cache.

Costs. Of course, as the latter example highlights, such independence is not free of cost: in the hash-table case, independent aggregations, for example, would later have to be *merged* by aggregating the aggregates themselves. Figure 2.3 depicts this more generally for all kinds of *split-merge* parallelism: if blocks of input are processed separately, any operation that re-orders records based on their values will require a subsequent merge–so called because of its resemblance to the eponymous step of mergesort–to stitch the results back together. In the paradigmatic plan of Figure 2.2b, all that follows the initial hash-table build may be split



(a) Spatial Parallelism

(b) Temporal Parallelism

Figure 2.4: Spatial and Temporal Parallelism

without a merge; splitting the hash-table build would necessitate a merge of the parallel tables.

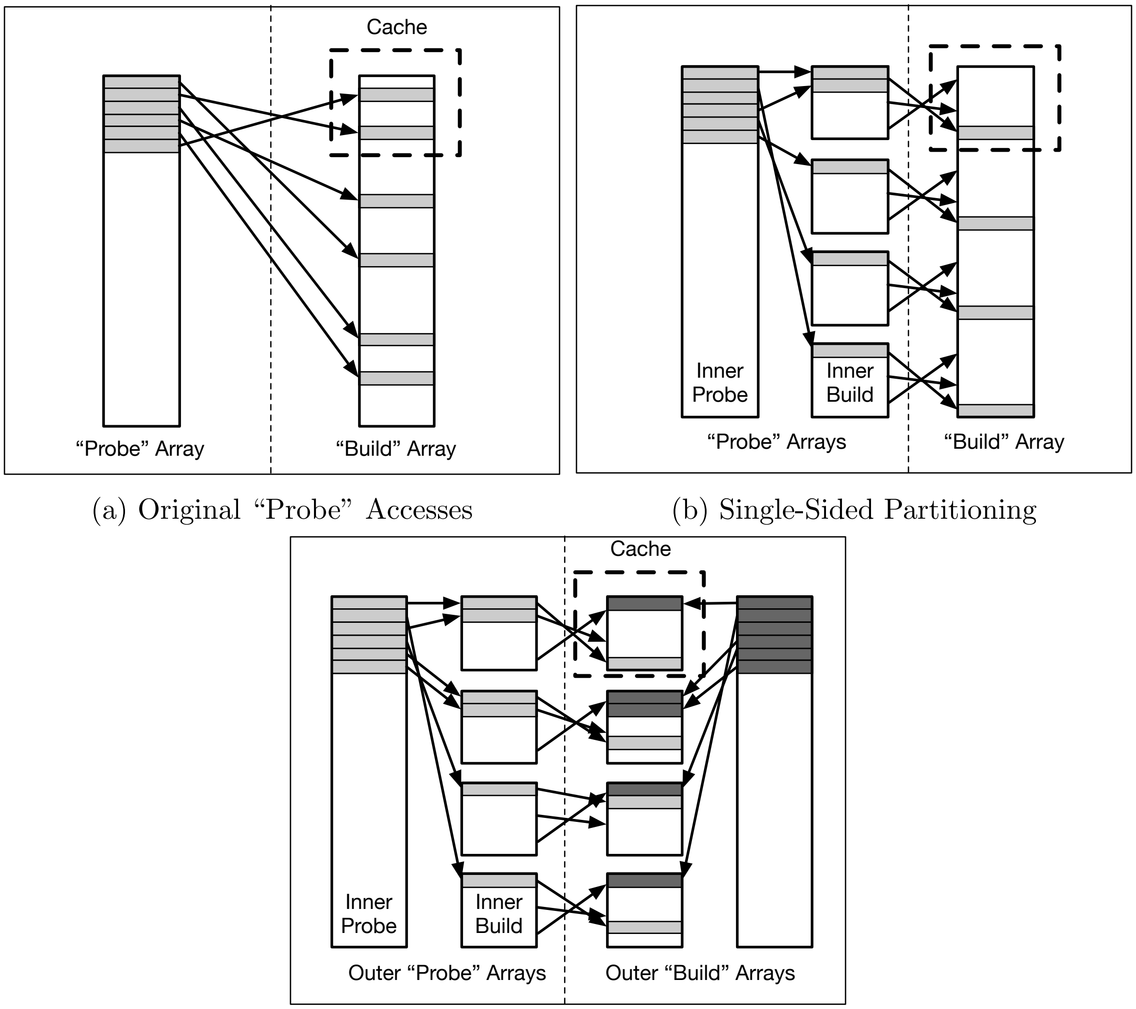
The impact of the split-merge transformation on performance will generally depend on the relative cost of this merger compared to the benefit of independence. If the data in each block are sufficiently re-used, then splitting may reduce traffic enough to improve performance even without the benefit of parallelism. On the other hand, even parallelism may be of minimal benefit if the merger must be performed serially.

The other possible cost of split-merger is that spatial parallelization sometimes requires data structures to be replicated once per “thread” (if atomic operations are not available or are too slow), thus increasing the size of the working set.

#2: Partition Parallelism

If split-merger is limited by the (possible) need to perform an expensive, serial merge operation, then its alternative is *partitioning*, which divides inputs among several partitions based on the values of some attribute chosen as the *key*, which is often a hash function of an underlying attribute used as a join or aggregation key. In radix partitioning, a subset of the key's bits are used as the address of each record's output partition, making the partitions independent, because a single key can appear in at most one of them. This value independence makes subsequent merging unnecessary. The parallelism induced by partitioning, like that from splitting, can also be exploited either spatially or temporally.

In the recurring example of Figure 2.2b's join, partitioning could apply to either the left (`lineitem`) side, the right (`parts`) side, or both. Figure 2.5 shows the consequences of these strategies in terms of memory access patterns. In the original non-partitioned plan, there are two operations that resemble Figure 2.5a: (1.) the initial construction of the hash-table



(a) Original “Probe” Accesses

(b) Single-Sided Partitioning

(c) Double-Sided Partitioning

Figure 2.5: Aggregation, Joins, and Hash Tables with and without Partitioning

of `part` on key field `p_partkey`, and (2.) the join loop, in which that hash-table is probed with records from `lineitem` using the `l_partkey` field as a key. In both cases, the physical array containing the hash-table acts as the *build* array, while data values in the two key fields determine which locations in that array are accessed. This means such accesses are possibly scattered across the length of the build array (depending on the distribution of the keys), generating many cache misses if it array is larger than the cache. This is a performance impediment not only on contemporary general-purpose processors, which can sustain only a limited number of outstanding misses at a time, but also by the more abstract metric of raw communication cost, if the width (in bytes) of the attribute fields accessed is much smaller than the size of a cache line.

Benefits. Partitioning can both reduce the likelihood of such cache misses, and provide additional parallelism. In Figure 2.5b, records are first distributed to independent buckets based on the addresses in the (outer) build array (i.e. keys) to which they will generate accesses, meaning that those resulting probes (arrows across the vertical dotted line) will be distributed over narrower ranges, hopefully resulting in more reuse of records once they have been brought into the cache. It also means that each of the resulting partitions can be processed in parallel, without atomics or locking, as they are guaranteed to access disjoint locations in memory (which matters in the aggregation or hash-table build case, though not in a join). In the case of a join, *double-sided* partitioning (Figure 2.5c) occurs when the hash-table build is parallelized by partitioning with the same function used to partition the probe relation, yielding in a miniature hash join between each corresponding pair of partitions in the two relations.

Costs. Partitioning requires both extra computation, and extra communication: in radix partitioning, the key fields of all input records must be first scanned once while a *histogram* is built to count the number of elements in each resulting partition, so that sufficient space in the output buffer can be allocated for them; then, all input records must be scanned again and scattered to their respective partitions, all while the histogram's counters must be incremented to track the offset at which the next record should be emitted. Thus, the total traffic cost is at least (1.) two full scans of the input keys, (2.) one full scan of input values, and (3.) one full (scattered) write of the output.

However, cache misses from random accesses may also occur: radix-m partitioning has a working set of at least 2^m cache lines–one per partition–since the next record to be output at any point could fall into any one, in the worst case of a uniform key distribution. The histogram, too, is part of the working set, though smaller, as one cache line is likely to hold several counters. In the worst case, a high-fanout partitioning may result in the transfer (read-modify-write) of one cache line per record during histogram-building and *two* during movement, in addition to the two full sequential scans of the input.

On realistic platforms, other non-bandwidth bottlenecks may emerge: cache misses between levels of the on-chip hierarchy may degrade performance of general-purpose pipelines even if no off-chip traffic is incurred, and other real-world constraints like cache conflicts and the reach of translation lookaside buffers (TLBs) may impede execution if the working set's virtual address range exceeds TLB capacity (though there are workarounds for this, such as huge pages, or software write-combining buffers [57] – see Section 3.1).

In order for partitioning to be profitable under a raw communication metric, it must eliminate more cache line transfers in a future join or aggregation than it causes in its own execution. If computation is considered, then any traffic increase must be offset by the resulting parallelism downstream in the query, which often may only be realized by way of partitioning.

#3: Attribute Packing & Memory Layout

A classical element of schema design in relational databases is *denormalization*, which eliminates copies of an attribute that is shared between two or more relations, relegating its storage instead to a single table. For example, in the TPC-H schema, a lineitem of an order pertains to a specific part with qualities, such as price, that are common to all orders in which it might be included. Instead of replicating that price for each lineitem in which the part is referenced, a separate `part` table includes a single copy of it, and so computing the cost of an order requires joining against the `part` table to retrieve it. This has a clear trade-off: it reduces the amount of storage required for the whole dataset, at the cost of additional joins.

Denormalization has an analogue in machine-level planing. Figure 2.2b illustrates the execution of precisely the join described above. The coarse schematic depicts a hash join using a table with two columns, which naturally correspond to the join key (the part number, encoded as `l_partkey` or `p_partkey` in `lineitem` and `part` respectively), and a *row ID*, which points back to each part's original location in the `part` table. When the join is performed, a matching part for a given partkey has other attributes that can be retrieved by *gathering* them from the original `part` table based on their row IDs. However, this need not be so. An alternative join style, such as shown in Figure 2.11b, could instead *pack* all attributes of the `part` relation into the hash-table, eliminating the need for any future gather. The same principle applies to partitioning as well, or any other shuffling operation, including sorting: when part of a record is moved based on one attribute, the others can either follow it or remain in place. The choice is not all-or-nothing, either: any subset of possibly-relevant attributes can be packed or not.

Benefits. Because hash-table construction (most likely) reorders entries significantly, and randomly, with respect to their original sequence, post-join gather operations will be scattered randomly over the original table, resulting in poor memory locality and a higher number of cache misses, which packing can eliminate. Any matching probe to a packed table can access all relevant attributes at once, in a single cache line, or at least a smaller number of them.

The decision to pack also offers an opportunity to adjust the memory layout of the reordered table. Just as any table can be arranged in row- or column-major order, so too can any packed hash-table. While it may seem counterintuitive to lay out packed hash-table attributes in discontiguous arrays, this can be fruitful when, e.g. the join key is itself rather selective, or one other attribute is used in a selective post-join filter, prior to others: in this case, all attributes accessed after the initial join-filter can be co-located in a single array that forms an effective secondary working set. Of course, such re-shaping of relations need not be incorporated into a hash-table build or partition, and can be applied on its own prior to the execution of any other operation.

Costs. In addition to the sequential read traffic required to scan all packed attributes, the addition of columns to a hash-table or partition output increases the size of the working set during its construction and, for hash-tables, all subsequent probes. If that increase exceeds cache capacity, the resulting capacity misses may dwarf any savings from reduced gathers.

```c
size_t count = 0;
for (size_t i = 0; i < N; i++) {
  bool a = A[i], b = B[i]; c = C[i];


  rec_t d = D[i];
  bool mask = a && b && c
  if (mask) out[count++] = d;

}
```

Listing (2.2) Complex predicate filter

```c
size_t count = 0;
for (size_t i = 0; i < N; i++) {
  bool a = A[i];
  if (a) {
    bool b = B[i], c = C[i];
    rec_t d = D[i];
    bool mask = b && c;
    if (mask) out[count++] = d;
  }
}
```

Listing (2.3) A “cracked” version of 2.2

Figure 2.6: C Code for Cracked Predicate Application

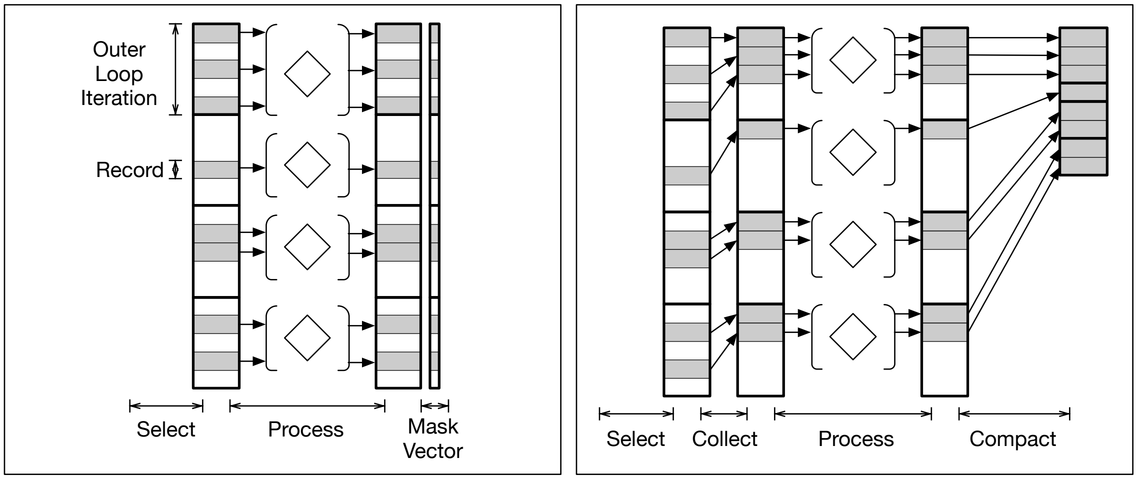
#4: Predicate Attribute “Cracking” & Access Serialization

Where attribute packing is not useful, its antithesis may well be. If a filter operator’s predicate consists of several disjunctive clauses, each of which uses a different attribute, then loading all fields may consume more bandwidth than necessary if initial clauses are highly selective. The canonical filter loop in Listing 2.2 exhibits precisely this excess. To its right, Listing 2.3 wraps accesses to the secondary attributes `B[]` and `C[]` (in separate columns) inside an `if` statement conditioned on the truth value of `A[]`.

The literature [64] has treated this distinction as one between *vectorization* (Listing 2.2) and *compilation* (Listing 2.3), insofar as the word ‘vectorization’ implies computing the full predicate for a column of records at a time without any intervening branches, while ‘compilation’ implies the opposite. A more generic term used in this thesis is *cracking*, which emphasizes the splitting up of predicate evaluation (and avoids conflicting meanings of ‘vectorization’, hereafter reserved for the exploitation of data-parallel architectures).

The cracking transformation encompasses not just the serialization of attribute accesses shown in Listing 2.3 but also any changes to memory layout and execution that result from choosing whether or not to coalesce the sparse subset of records that survive a filter after it is applied and before the next materialization occurs. For example, both main loops in Figure 2.6 build a dense array of valid records in their output, but these loops could be split into two, with a vector-valued `mask` output in between, as in Figure 2.7a.

Or, the loop could be split into two *with* coalescing in between (“collect” in Figure 2.7b), producing dense runs of elements of `A`, which may then be processed without any mask. In this case, “processing” would entail gathering needed elements from `B[]`, `C[]`, and `D[]` via position values stored during collection. This is intermediate buffering is sometimes necessary in real systems to obtain any benefit from the communication reduction of cracking, as Polychroniou [48] and others have shown. Normally, this buffering would occur on a block-wise basis: by applying temporal splitting in conjunction with cracking, an idealized query plan assembles a block of valid addresses to gather in the next loop that is sufficiently large to amortize any startup overhead while also small enough not to exhaust cache capacity and so have to spill to memory between iterations of the outer loop.



(a) Masked Execution

(b) Collection & Compaction

Figure 2.7: Masking vs. Collecting & Compacting

Benefits. Under a pure communication cost metric, cracking is beneficial when the selectivity of each initial predicate is low enough to totally eliminate the need for access to at least some cache lines containing other attributes. Whether this is the case depends on both the selectivity and the size of the attribute(s) being accessed, as Pirk et al. [45] show: if the cache line is larger than the attribute size (e.g. eight 8-byte values in a 64-byte line), then a 50% selectivity will have for example a negligible impact on the number of lines transferred, while an 8% selectivity might cut communication roughly in half. The benefit of coalescing is that fewer cache lines may be needed to contain a materialized output if a denser packing of valid records causes fewer lines to be touched.

Under a cost metric that also considers computational requirements, cracking can eliminate superfluous expression evaluation in addition to suppressing unneeded loads and stores. Meanwhile, coalescing obviates the need to access mask or “valid” bits for each tuple, removing conditional branches.

Costs. In terms of memory traffic, cracking should have a worst-case neutral cost when considered in isolation. However, it conflicts with attribute packing, which negates its benefit by putting separate fields in adjacent memory and thus within the spatial footprint of a single line transfer. Coalescing can potentially increase memory traffic if it forces the materialization of intermediate results where none otherwise was needed.

In the real world of general-purpose CPUs, cracking entails additional data-dependent branches, which can dramatically decrease performance when their selectivity nears 50% and their outcomes therefore become unpredictable, causing frequent full or partial pipeline flushes (or other control-flow divergence phenomena on data-parallel processors), as others have reported [64].

#5: Dynamic Arrays vs. Pre-Allocation

Common operations such as joins and hash-table builds produce results whose sizes are not statically knowable in advance, or which are divided into partitions or buckets of initial unknown lengths. [1]

Even if a sophisticated query optimizer estimates join cardinality with high accuracy (though likely it will not [33]), this guess may still be exceeded at runtime, requiring dynamic allocation of additional space in necessarily disjoint memory. This discontinuity will affect all consumers of the operator's result, which must traverse all extents of records allocated in the course of execution. Eliminating that discontinuity requires *compacting* all such extents, which can be done at any downstream point in the query plan.

Alternatively, the inputs to operators with dynamically-sized output can be processed in two passes: one to *count* the number of outputs that will be generated (or the sizes of each of its buckets or partitions), and one to perform the operation itself. This is almost universally the case for partitioning, but could be applied to joins as well. Hash tables are a trickier case, as it is impossible to count in advance the number of distinct keys falling into each bucket without actually enumerating them, which requires an allocation of space equal to the number of keys. Still, it is possible to at least tabulate an upper bound on the number of keys that map to each bucket, which will over-provision space in the case that many elements share a key and are aggregated, but at least guarantees that the total size of all buckets combined is no larger than the original input, rather than its square, as would be needed to pre-allocate without such a count.

Benefits. Pre-allocation eliminates the need for linked list traversals, unpredictable dynamic allocation, and compaction, and can drastically reduce the need to over-provision hash-table and partition space in the presence of non-uniform key distributions and imperfect hash functions. Under a strict communication metric, choosing to pre-allocate may not have a noticeable salutary effect, but its elimination of irregular branches, serialized memory accesses, and dynamic allocation may significantly reduce computational bottlenecks on real systems.

Costs. The traffic cost of pre-allocation is an additional sequential scan of all input records' keys, which may be substantial. If the keys themselves are derived from other attributes, then these may have to be re-scanned as well, or else the computed keys must be written during the pre-allocation pass and then re-read during the table build or partition phase. (This is the same as in partitioning.)

[1] Traditionally, hash-tables are built either with *chaining*, in which each bucket contains an expansible linked list of elements, or *linear probing*, in which a contiguous array of buckets is allocated and entries are inserted at the first open slot after their buckets' beginning, and the whole table is re-allocated and re-built if its space is exhausted. For the sake of brevity we forego consideration of the latter.

#6: Operator Fusion & Interleaving

A primary benefit of query compilation is the opportunity to *fuse* together multiple physical operators into a single compiled unit. Traditionally, this meant identifying chains of dependent operators terminated by a *blocking* operator, which is one that must fully consume its inputs before any of its output may be used (e.g. in aggregations and hash-table builds). Each record produced by one operator in the chain may be consumed by the next without necessarily being written out to memory, resulting in a single loop over the inputs to the first operator. All intermediate results simply live in registers.

The various forms of parallelism described above necessitate an extension of this paradigm to include the fusion, at another level, of even some operators that *do* forcibly spill to memory: when a blocking operator such as aggregation processes input that has been parallelized (in space or time), it need only spill the contents of its local input fragment before a subsequent consumer uses it, as the parallelization transformation must, for correctness, have included a later merge operation. If parallelization is conceived in terms parallel loops, then this form of fusion can be said to act on *outer loops*, with intermediate products confined, if not to registers, then at least to smaller regions of memory, effectively *interleaving* execution of outer-loop-fused operators at the granularity of a block of records. And if special operators exist to synchronize their parallel brethren, then these are outer-loop fusion's analogues to inner-loop fusion blockers (Chapters 3 and 4 supply precise examples).

In either case, the mere presence of blocking operations is not enough to determine completely the way in which a parallel query plan must be divided into discrete loop nests. In Figure 2.8, for instance, which shows a plan for some unstated query for the sake of example, the blocking hash-table build merely precludes its own inclusion in a single (inner) loop with the downstream join. The rest of the query must also be divided between these two loops, but cutting the plan along any of the dotted lines (at least) would result in a valid split. Each one may have communication requirements different from the others, depending on the size of intermediate results to be materialized. In machine-level planning, such alternatives must be considered as part of the query's algorithmic design space.

A somewhat surprising aspect of Figure 2.8's plan is that it is not tree-structured, and contains a cycle in the underlying undirected graph. This reflects the possibility that a single operator's result may be shared among multiple consumers, all of which may or may not be fused into the same loop. In this case, a computation (`Eval`: "evaluate" an expression) produces an output that may need to be materialized if any of cuts (1), (2), or (3) is drawn. However, an alternative is possible (Figure 2.9): that node could simply be *repeated* in two separate loops to avoid materializing its result, thus potentially trading off extra computation for reduced communication, though not necessarily.

Benefits. Fusion can dramatically reduce memory traffic by eliminating the materialization of large intermediate results, and replication of computation can potentially do the same. In computational terms, the fusion also removes any load and store operations necessary to write out and then re-scan operators' outputs, while inner-loop fusion also allows compilers to optimize operator code across erstwhile loop boundaries.

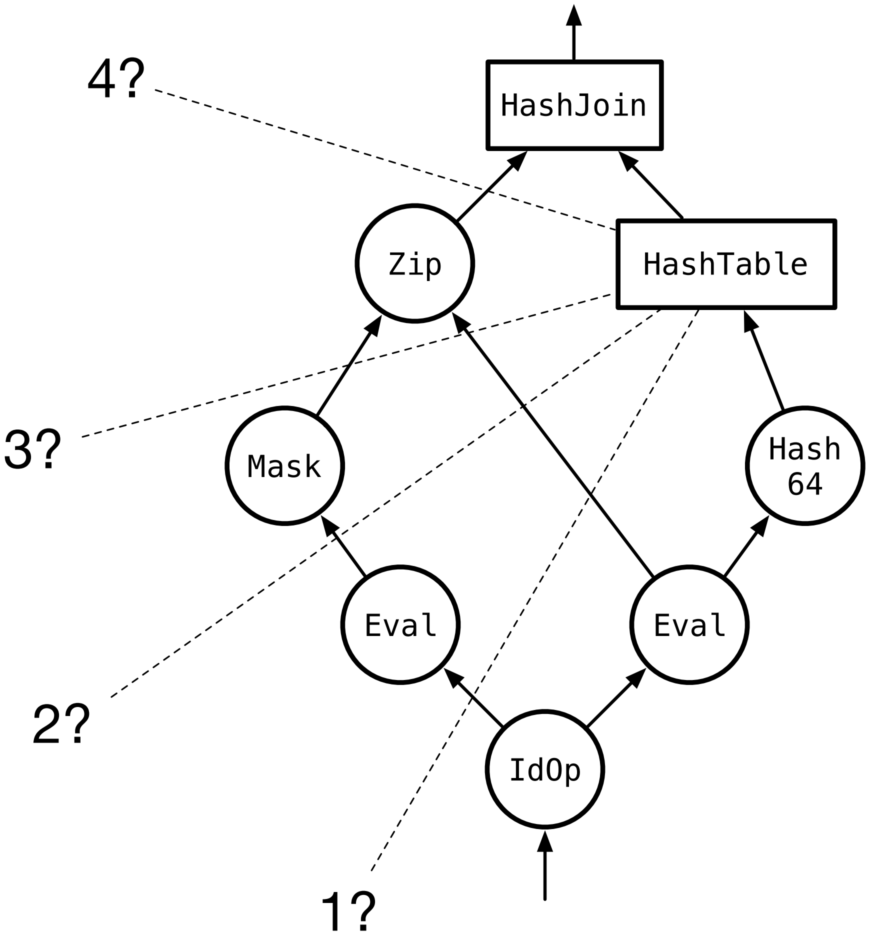


Figure 2.8: Where to "cut" inner-loop fusion? HiRes allows a single scan to drive an operator that produces multiple results, and enables the choice of multiple such fusions, indicated by dotted lines.

Costs. Even under a strict communication metric, the costs of different fusion choices can vary wildly depending on the degree to which operators on different loop boundaries change the size of their outputs, making the DAG cut problem an important optimization parameter. Additionally, the inclusion of more operators into a fused loop increases the working set of that loop to the sum of working sets of its components, potentially overflowing available cache capacity and thus generating a large increase in traffic. For example, two fused or interleaved hash joins both add their hash-tables to the fused loop's working set, which very likely represents a large expansion.

Even the attempted tradeoff of extra computation for reduced traffic in Figure 2.9 can have the opposite of its intended effect. This happens when, for instance, its input was already needed–and therefore read by–another operator in the original loop from which it

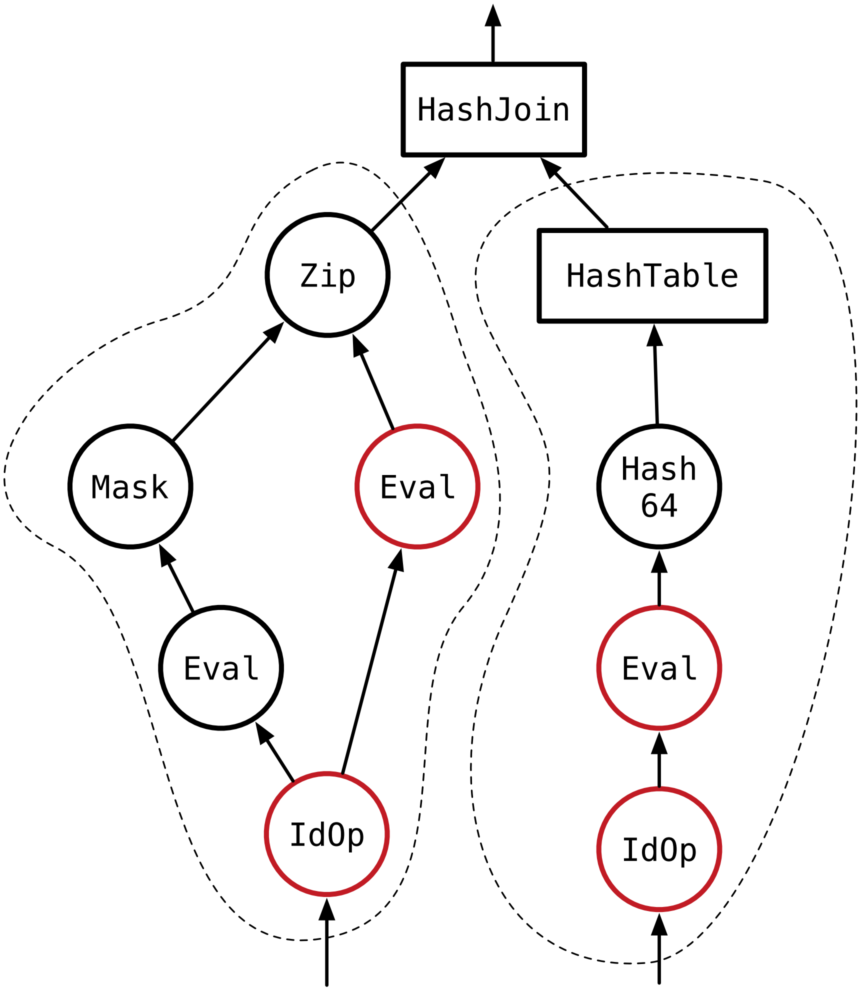


Figure 2.9: The "one output per pipeline" model. Following the query DAG model of HyPer [25] and Hawk [9], this plan first would result from traversing the left side of the join expression until finding a materializer, to produce one "pipeline", and then doing the same for the right. Each fused loop has exactly one output, and the computation (including scans) in red is repeated.

was copied, causing its clone to repeat the same scan of records, which may be of greater size than its output (e.g. an arithmetic expression that adds four integer attributes to produce a single integer result).

Mere fusion should not impose any additional computation costs, but in the presence of multi-level memory hierarchies, an increase in working set size that does not result in traffic to off-chip DRAM may nevertheless cause misses within the upper, on-chip levels of cache that realistic micro-architectures cannot completely hide. Computational replication, of course, poses a more direct arithmetic cost.

Summary of MLP Transformations

Transformation		Benefits	Costs
Split-Merge Parallel.	**M** **C**	Cache blocking (temporal) Parallel processing (spatial)	Final merger
Partition Parallel.	**M** **C**	Cache Blocking (tmp+spa) Parallelism (spatial)	Multi-pass, scatter Hist. build
Field Packing	**M**	No gathers	Bigger working set, scans
Col.-Major Build	**M**	Allow cracking within table	Excludes packing
Predicate "Cracking"	**M** **C**	Avoid unneeded access	Packing precludes Data-dependent branches
Mask Compaction	**M** **C**	Smaller mat. output No mask branch	Forced materialization Less data-parallel
Pre-Allocation	**M** **C**	Smaller output No pointer chase	Scan keys twice Re-compute keys
Operator Fusion	**M**	Intermediates in registers/cache	Combine working sets

Figure 2.10: Summary of MLP Transformations. (**M** = Memory Traffic, **C** = Compute)

2.4 Case Study: TPC-H Query 19

To illustrate the space of query implementations defined by MLP transformations, and to show how such choices interact across the across the totality of a query plan, we return to the running example of Query 19 from the TPC-H benchmark suite.

Packing & Partitioning

Figure 2.11 shows three machine-level plans derived from the baseline plan in Figure 2.2b taken from the combinatorial space:

$$\mathcal{S} = \left\{\{\text{partition}\} \times 2^{\{\text{left,right}\}}\right\} \times \left\{\{\text{pack}\} \times 2^{\{\text{left,right}\}}\right\}$$

This set $\mathcal{S}$ contains at least $|\mathcal{S}| = 4 \times 4 = 16$ plans, and more if "left" and "right" are allowed as shorthands for the sets of all subsets of their attributes. Meanwhile, "partition" can be expanded with a radix parameter, and all plans can be crossed with the set of possible hash functions on the key, including "none" (giving an *array join* in the terminology of Schuh et al.[59], where the values of the keys themselves become indices into a hash-table array). In reality, the space is even larger, because even the "non-packed" plans can have attributes packed into their relations' partitioned forms, if any are used.

The diagrams in Figure 2.11 show, in particular, the execution flow of plans where both relations are either partitioned, packed, or both. In Sub-Figure 2.11a, partitioning (the key attribute of) both relations results in (1.) probes that are constrained within partition-sized–and hopefully cache-sized–subsets of the hash-table, (2.) a hash-table build that can occur in parallel for each partition, which is independent of all others, (3.) an automatic parallelization of the join, due to the independence of each pair of corresponding partitions in the build and probe relations, and (4.) the automatic partition-wise parallelization of all the comes after the join as well. In 2.11b, all relevant attributes of both relations are scanned, and packed into contiguous arrays before the join, entirely eliminating the need for any subsequent gathers, which might otherwise have entailed random accesses to several cache lines per tuple. Finally, the far right panel depicts a plan in which the parallelism and locality features of the left are combined with the communication-saving properties of the middle.

It is clear *a priori* that not all plans in $\mathcal{S}$ are sensible: even Figure 2.11b includes a "pack" of the build relation, which would be more obviously beneficial if it were re-used, which it is not. However, these plans are considered here because exposing the broadest possible space of implementations is a prerequisite to uncovering the unlikely winners, and the translation of human insight into a useful cost model will take serious effort.

Parallelization

The independent probes depicted in the fully-partitioned plan of Figure 2.11 represent one form of query parallelization, in which the corresponding partitions of `lineitem` and `part` are distributed across processing elements. Many other parallelizations are possible. Partition-parallelism may be combined with split-merge parallelism recursively, with the results of each treated either spatially or temporally.

This paradigmatic single-join query can be parallelized in two phases: the hash-table build, and then the probe that completes the join. They offer separate, but not completely independent choices, as the structure of the hash-table(s) impact how the probes may be parallelized, so we consider that first.

Build Phase

It is worth noting that one plausible build strategy is simply to build the table serially, especially if the build relation is significantly smaller than the probe relation, or becomes so after relevant filters are applied. Though a large size disparity does exist between the actual relations implicated in Query 19, no appropriate predicates shrink the table size, and serialization of the build phase has proven a bottleneck in practice ().

Split-merge parallelization is not likely to prove fruitful here, as the merger would have to be performed serially, negating any benefit of (spatial) splitting. This strategy is generally profitable only when each parallel build reduces the size of its input dramatically, such as when aggregation is applied as part of the build, but that is not the case in this query. Instead, parallelization by partitioning is likely to be more advantageous, and the resulting independent hash-tables can then be treated separately by the build phase, or as one single hash-table that simply took less time to create.

Probe Phase

Regardless of how the hash-table build is parallelized, the probe phase can simply be split spatially, with merger needed only when the resulting parallelism extends across the join through some other operations (such as aggregation or group-by) that themselves require it (in this example query, the merge is a reduction by sum of each thread's final total).

This spatial splitting can combine with partitioning, once again independently of any partitioning used in the build phase (using a different radix/number of partitions), and with temporal splitting too. For example, one strategy entails splitting input records spatially among threads, each of which partitions its separate domain. Then, each thread of execution may join each of its local partitions in sequence, achieving a greater degree of cache locality. Furthermore, if the distribution of keys is very uniform, and all threads share a common outer-level cache, then they will likely share cached regions of the hash-table as they target overlapping ranges of its buckets. On multi-socket systems, spatial splitting prior to partitioning confines each thread's scattered writes during the record move phase to be within its own NUMA domain, potentially ameliorating otherwise problematic cross-socket write traffic.

An alternative, **part-split-build**, inverts the preceding by performing parallel partitioning to a set of shared partition buffers, each of which is then processed in sequence but parallelized across all threads. The entire probe can also be wrapped in a temporal split that processes one last-level cache-sized block at a time, meaning that partitioning is cache-blocked at a higher level of the memory hierarchy.

Masking & Filtering

A classical query optimizer will surely preface the join with selection on the `l_shipinstruct` and `l_shipmode` attributes, as Figure 2.2a indicates, but fusion and parallelization cause the masking and cracking choices of record selection to "interact" with the choice of join strategy. If no partitioning is employed, then input records can simply be masked, preventing deselected records from participating in the probe. However, parallelization of the probe phase requires that the selection be parallelized in precisely the same manner, or else it could not be fused into the same (inner) loop as the probe, necessitating materialization. Thus, *parallelization of the join must occur at the start of the whole plan.*

In this particular class of query plans, no explicit mask compaction is necessary, as the join is "naturally" compacting – the independence between the its input and output sizes already packs successful probes densely, and records that fail the prior selection simply act as if they failed to find a match. However, this does not preclude the insertion of explicit compaction inside the filtering process, as may be desired when it is cracked.

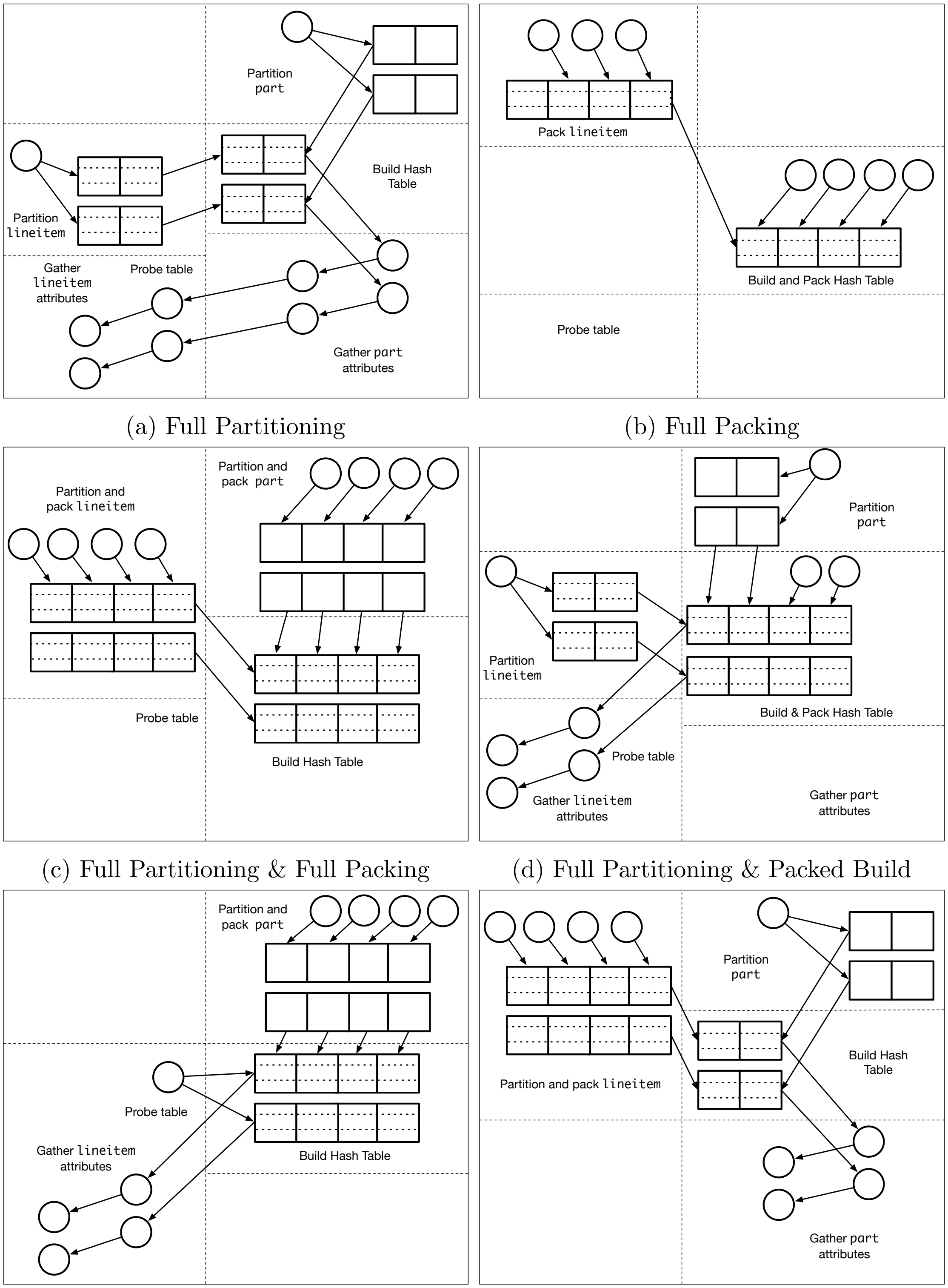
Fusion Choices

If radix partitioning is used as part of the probe strategy, then selection ought to precede it, as it will reduce the number of records to be moved. Since partitioning has an extra initial loop to build a histogram, selection must fuse with that if it is to fuse at all, leaving a choice of what, if anything, to materialize in the process. If selection is mere masking, then it is possible simply to output nothing, in which case all attributes used to compute the selection predicate must be re-scanned during the move phase of partitioning, and any arithmetic expressions must be re-computed.

Alternatively, the mask itself – one bit per record – can be materialized during histogram building, and re-read during movement. Then, it can be used to skip accesses to input records whose bit is not set, or not, if the selectivity is close to 50% and this proves counterproductive. In the latter case, compaction may well be worth fusing into the histogram build.

Conclusion

Many of the plan parameters described above result in *fundamentally different code*, both in terms of the bodies of column-processing loops, and the set of loops themselves. As such, Chapter 3 describes a language and compiler stack for expressing machine-level plans, and generating their respective programs as explicitly parallel loop nests.



(a) Full Partitioning

(b) Full Packing

(c) Full Partitioning & Full Packing

(d) Full Partitioning & Packed Build

(e) Partitioned & Packed Build

(f) Full Partitioning & Packed Probe

Figure 2.11: Partitioning and Packing

Chapter 3

HiRes: A Language for MLP

We introduce a language called HiRes whose aim is to expose the six query planning dimensions described in Section 2. It is an array-processing language, where, as in predecessors such as NESL [6], parallelism is expressed by the depth of array nesting imposed on otherwise flat data. Unlike NESL, its execution is statically scheduled, and it needs no runtime system (beyond OpenMP's) to interpret this structure. HiRes's constructs give the programmer (or DBMS, or auto-tuner) control over *loop nests*, by means of nesting depth and fusion decisions, and *memory layout*, by way of its type system, which encodes row- vs. column-major storage, predicate masks (null bits), fixed- vs. variable-length array dimensions, and contiguous vs. disjoint buffers, all within an arbitrary array nest.

HiRes is a Scala-embedded domain-specific language (DSL) for expressing machine-level plans and whose prototype HiRes-compiler emits C++11 code with parallel loop annotations[1] suitable for compilation with any standard toolchain[2]. At the moment, it has no front-end to parse and algebrize SQL programs, or perform more classical kinds of query optimization, so we require queries to be input as HiRes code directly. This also bespeaks the suitability of HiRes as a framework for building more general kinds of collections libraries for use outside the domain of pure database analytics.

Between HiRes and C++, a C-like intermediate form called LoRes is used to represent queries as concrete loop nests, and enable several low-level transformation passes. The more interesting aspects of that process are described at greater length in Chapter 5. The rest of this chapter, meanwhile, gives an introduction to the HiRes language, surveys its primitive constructs, and explains how its type system supports the loop nest and memory layout controls evoked above. The conclusion illustrates how to construct and optimize query plans with HiRes.

[1] In future work, we hope to expand this to include SIMD and vector instructions as well.

[2] Unlike prior work [25], our research prototype does not refine this further to LLVM's [31] intermediate language, as we have not yet been concerned with compilation time. There is no fundamental reason it could not do so in the future, though of course the overhead exporting LLVM IR from the JVM may prove to be a bottleneck.

3.1 HiRes Operators & Syntax

To introduce the language, this section presents short snippets of HiRes code as they would actually be written in the host language, Scala, and highlights their use of syntactic sugar where appropriate. Operators' types (whose meanings will be explained more fully in Section 3.2) are stated in a symbolic form, since their concrete realizations as Scala values inside the HiRes compiler are too unwieldy for the printed page, and would often span multiple lines of text. They rarely need be spelled out inside HiRes programs, however, as the language's implicit typing rules render such verbosity unnecessary. Type annotations such as `scan1: Operator` in the examples are thus *Scala* types, indicating in this case that an identifier refers to a HiRes `Operator` AST.

All code listings are examples of *meta-programming* to various degrees: Scala constructs (`val`, `var`, `=`, `if`, `for`, etc.) are used to assemble HiRes ASTs programmatically. It is beyond the scope of this thesis to provide a tutorial of the Scala language [42], but it is usually possible to distinguish its keywords from those of HiRes, as the latter are capitalized.

HiRes Programs as Operator Compositions

A HiRes program is a composition of *operators*, each of which has the type of a *function*, usually from one kind of vector of records to another. Figure 3.2 summarizes all the primitive operators out of which compound HiRes programs are built. A guiding principle is that each primitive should translate to *one* loop nest over its input.

This scan-selection operator is a "Hello World" example:

```
1 val scan1: Operator = {
2   Let(List(
3     'cond := 'input('value < 'maxval),
4     'mask := Mask('input, 'cond)
5   ),
6   Collect('mask.project('key -> 'key, 'value -> 'value)))
7 }
```

Listing 3.1: A simple HiRes scan-filter

The above operator selects all records whose `value` field is less than a threshold, and might have the type:

$$(\mathsf{V}\left[((\texttt{key} : \mathsf{Index}), (\texttt{value} : \mathsf{Int}))\right], \mathsf{Int}) \rightarrow \bar{\mathsf{V}}\left[(\texttt{key} : \mathsf{Index}, \texttt{value} : \mathsf{Int})\right]$$

This signature depends on the surrounding context (i.e. the original type of `'input`), as described below in 3.1, and $\mathsf{V}[X]$ is read as "a vector of X".

On line 3, the `'input(...)` term is syntactic sugar for

```
Eval(IdOp(Id("input")), Id("value") < Id("maxval"))
```

The single quote embeds a HiRes symbol in Scala, while the parentheses stand for an `Eval()` operator that computes an expression for each record of the input. Inside the `Eval()` statement, the right-hand side is a scalar, primitive-valued expression where each identifier refers, implicitly, to the name of a field in the record type of each element in the left-hand side input. Thus, the above expression is well-typed because we have assumed that the record contains a numeric-typed field named `value`. By itself, the evaluation has a function type of

$$\mathsf{V}\left[\left(\left(\mathtt{key} : \mathsf{Index}\right), \left(\mathtt{value} : \mathsf{Int}\right)\right)\right] \to \mathsf{V}\left[\mathsf{Bool}\right]$$

In this case, the boolean result type is inferred from the value of the expression, for which the vector structure is copied from the left-hand input.

On line 4, the `Mask()` operator binds `'cond` as a predicate mask for the rows of `'input`, yielding a result in `'mask` of type $\mathsf{V}^{P}\left[\left(\left(\mathtt{key} : \mathsf{Index}\right), \left(\mathtt{value} : \mathsf{Int}\right)\right)\right]$. Unlike `Eval()`, the `Mask()` operator does not actually produce any new data, but merely *associates* the existing boolean vector `'cond` with the existing records of `'input` (which can still be accessed unmasked under that name). `'cond` may never even exist as a physical vector in main memory, since inner loop fusion (Section 4.3) of the whole program will avoid materializing it. The final `Collect()` operator removes the predicate mask, by packing valid elements contiguously and converting the array type to one with a num-valid marker $\bar{\mathsf{V}}\left[\left(\ldots\right)\right]$. It is a *materializing* operator, meaning that its output must be written out to memory, and that it must finish fully before a subsequent operator may use its results.

Line 6 desugars to a combination of `Let()` and `Project()`, which must be understood together:

```
Let(List(
    'key := Eval('mask, 'key),
    'value := Eval('mask, 'value)
  ),
  Project('key, 'value))
```

A projection works in tandem with a `Let()` to create a *new flat record* type whose fields bear the names of the identifiers to which its non-record inputs are bound; any actual records in included in the input will have their own fields' names reproduced in the result type. The counterpart to `Project()` is `Zip()`, which creates a new split record type with fields that reside in physically disjoint columns in memory. This combination is common enough to warrant its own abbreviation in the form of the `Operator` class' `.project()` and `.zip()` methods.

Inside `Let()`, the final constituent determines the output type and semantics of the whole, using any bindings brought in scope by the assignment list above it, or which are in scope due to an outer let statement.

Using HiRes Programs

Most expressions encountered thus far had implicitly inferred types, but compiling an operator requires at some point that concrete input and output types be specified to enable this inference, and to ensure that the implementation conforms to the desired specification. A `Func` specifier fulfills this expectation of the compiler's API:

```scala
val funcType = {
  val rec: Record =
    lo.SplitRecord(
      lo.Record.Field(lo.Index(), Some('key)) :: Nil,
      lo.Record.Field(lo.LoInt(), Some('value)) :: Nil)

  Func(
    Map[Id, hi.Data]('input -> Vec(rec), 'maxval -> lo.LoInt()),
    Vec(rec, numValid=true))
}
```

The preceding lines convey both the undesirability of stipulating every operator's type signature, as well as the utility of meta-programming to construct a complex type AST: the Scala `rec` variable allows the input and output types to share the same record structure.

Interfacing with the Outside World

Compilation of the last code fragment produces the header file shown in Figure 3.1. In the C++ output, all physical buffers are declared as `res_vec` vector types whose allocation routines are more efficient than those of the `std::vector<T>` class of the C++ Standard Template Library[3], but like the latter still store the size of the underlying buffer. Struct definitions for implicitly-defined record types are emitted as necessary, such as `_rss_0`, which contains the named fields of each record in the row-major-ordered output. Although the names of these types are not fully predictable (the `0` in `_rss_0` reflects the order in which it was generated relative to other types), the names of each struct's member fields can be, and the C++11 `auto` keyword can be used to capture the result of the operator's `_run()` method in code that interfaces with the auto-generated header file.

The resulting code is more verbose than what a human would write. This is due in part to aggressive LoRes AST-level common subexpression elimination (described in more detail in Chapter 5), which is necessary for some of the analyses the compiler was intended to facilitate, and also does help to mitigate some of the pathologies of auto-generated C++ code in other cases.

Nesting and Threading

HiRes expresses parallelism by increasing the degree of nesting of *arrays* on top of vectors. The kind of parallelism implied can be either spatial or temporal (or a combination): in either case, it really means an additional for-loop level around each (possibly fused) operator, and this may or may not be annotated with a parallel `#pragma`[4]. The twin `SplitPar` and

[3] This was found to be a significant bottleneck in earlier versions of the HiRes compiler

[4] Or equivalent construct in future targets other than OpenMP

```
struct __rec1_arr {
 res_vec<size_t>* key;
 res_vec<int>* value;
};

struct _rss_0 {
 size_t key;
 int value;
};

struct _lo_arr__rec0 {
 res_vec<struct _rss_0>* arr;
 size_t nval;
};


class Ressort_scan32indexint
{
 public:
  struct __rec1_arr* input;
  int maxval;
  Ressort_scan32indexint () {}

  struct _lo_arr__rec0* _run()
  {
   size_t t0;
   size_t t11;
   t0 = (*((input->key))).size();
   res_vec<struct _rss_0>* t_coll;
   t_coll =
    new res_vec<struct _rss_0>(t0);
   size_t t_nval;
   t_nval = t0;
   size_t ocur;
   ocur = 0;
   const size_t _CTMP1_fcur_max = t0;
   for(size_t fcur = 0;
     fcur<_CTMP1_fcur_max;
     fcur = fcur+(1))
   {
```

Listing 3.2: Sample C++ output

```
    size_t t3;
    int t4;
    bool t7;
    t3 = fcur;
    t4 = (*((input->value)))[t3];
    t11 = ocur;
    bool t_;
    t_ = t4<maxval;
    t7 = t_;
    size_t key_;
    if(t7)
    {
     size_t t8;
     t8 = (*((input->key)))[t3];
     key_ = t8;
    }
    int value_;
    if(t7)
    {
     value_ = t4;
    }
    int t_prj_value;
    size_t t_prj_key;
    if(t7)
    {
     t_prj_key = key_;
     t_prj_value = value_;
     ((*(t_coll))[t11]).value =
      t_prj_value;
     ((*(t_coll))[t11]).key =
      t_prj_key;
     ocur = t11+(1);
    }
    t_nval = ocur;
   }
   struct _lo_arr__rec0* out_wrap;
   out_wrap = new struct
  _lo_arr__rec0;
   out_wrap->arr = t_coll;
   out_wrap->nval = t_nval;
   return out_wrap;
  }
};
```

Listing 3.3: (continued)

Figure 3.1: C++ Output from Compilation of Listing 3.1

Operator	**Type Rule**	**Description**
Data-Parallel “Vector” Operators		
$\texttt{IdOp}(O)$	$\mathsf{V}[S] \to \mathsf{V}[S]$	Scan: contents of column O
$\texttt{Eval}(o,\ e)$	$(\mathsf{V}[S_1], S_1 \to S_2) \to \mathsf{V}[S_2]$	Compute an expression for each record
$\texttt{Position}(o)$	$\mathsf{V}[S] \to \mathsf{V}[\mathsf{Index}]$	Index of each rec. in array
$\texttt{Hash64}(o,\ n)$	$(\mathsf{V}[S], \mathsf{Int}) \to \mathsf{V}[\mathsf{Index}]$	64-bit hash (masked to n bits)
$\texttt{Gather}(o_{idx},\ o_{targ})$	$(\mathsf{V}[\mathsf{Index}], \mathsf{V}[S]) \to \mathsf{V}[S]$	Records at specified positions in o_{targ}
Shape-Changing and Nesting Operators		
$\texttt{SplitSeq}(o,\ n)$	$(\mathsf{V}[S], \mathsf{Int}) \to \mathsf{A}[\mathsf{V}[S]]$	Divide input into n equal-sized blocks
$\texttt{SplitPar}(o,\ n)$	$(\mathsf{V}[S], \mathsf{Int}) \to \mathsf{A}[\mathsf{V}[S]]$	Same as above, but blocks in parallel
$\texttt{Shell}(o)$	$(\mathsf{V}[S], \mathsf{Int}) \to \mathsf{A}[\mathsf{V}[S]]$	“Repeats” vector across all arrays
$\texttt{Flatten}(o)$	$(\mathsf{A})[T] \to T$	Removes (array) nesting
$\texttt{Project}(o_1, \ldots, o_n)$	$(\mathsf{V}[S_1], \ldots, \mathsf{V}[S_n]) \to$ $\mathsf{V}[(S_1, \ldots, S_n)]$	Pack all input types into a flat vector
$\texttt{Zip}(o_1, \ldots, o_n)$	$(\mathsf{V}[S_1], \ldots, \mathsf{V}[S_n]) \to$ $\mathsf{V}[((S_1), \ldots, (S_n))]$	Make rec. with multiple columns
$\texttt{Mask}(o,\ o_{cond})$	$(\mathsf{V}[S], \mathsf{V}[\mathsf{Bool}]) \to \mathsf{V}^P[S]$	Bind o_{cond} as a predicate mask to o
Histogram and Reduction Operators		
$\texttt{Histogram}(o,\ n)$	$\mathsf{V}[\mathsf{Int}] \to \mathsf{H}_{\text{built}}$	Counts each of the n keys in o
$\texttt{Offsets}(o, (n))$	$\mathsf{H}_{\text{built}} \to \mathsf{H}_{\text{off}}$	Prefix sum from counts to offsets
$\texttt{Reduce}(o, \odot)$	$\odot \in \{+, *, \max, \min, \|\|\}$ $\mathsf{V}[S] \to S$	Reduce a vector with op.
$\texttt{NestedReduce}(o, \odot)$	$\mathsf{A}[S] \to S$	Reduce an *array* with op.
Record Movement Operators		
$\texttt{Partition}(o_{key}, o_{val}, h)$	$(\mathsf{V}[\mathsf{Int}], \mathsf{V}[S], \mathsf{H}_{\text{off}}) \to$ $(\mathsf{V}[S], \mathsf{H}_{\text{end}})$	Partitions records based on their keys
$\texttt{Collect}(o)$	$\mathsf{V}^P[S] \to \bar{\mathsf{V}}[S]$, $\mathsf{A}\left[\mathsf{V}^P[S]\right] \to \bar{\mathsf{A}}[\mathsf{V}[s]]$	Remove non-masked elements
$\texttt{Compact}(o,\ h)$	$\left(\bar{\mathsf{A}}[\mathsf{V}[S]], \mathsf{H}_{\text{off}}\right) \to \mathsf{A}[\mathsf{V}[s]]$	Move recs. to elim. non-valids
$\texttt{InsertionSort}(o,\ k_1, \ldots)$	$\mathsf{V}[S] \to \mathsf{V}[S]$	Sort records by given key fields
Hash Table Operators (Description in text)		
$\texttt{HashTable}(o, \ldots)$	$(\mathsf{V}[S], \ldots) \to \mathsf{A}^C[\mathsf{V}[(\ldots)]]$	Build a hash table / aggregate
$\texttt{HashJoin}(o_L, o_R \ldots)$	$(\mathsf{V}[S_1], \mathsf{A}[\mathsf{V}[S_2]]) \to$ $\mathsf{A}^C[\mathsf{V}[(S_1, S_2)]]$	Join: use keys o_L to set array of o_R
Control & Declaration Operators		
$\texttt{Let}(o_1 \ldots o_n, \texttt{in = } o,\ \texttt{in })$	[See text]	Re-use nodes, create records
$\texttt{Cat}(o_1 \ldots o_n)$	$((t_1), \ldots, (t_n)) \to (t_1, \ldots, t_n)$	Combine all nodes’ outputs
$\texttt{Uncat}(o, N)$	$(t_1, \ldots, t_n) \to t_N$	Extracts the Nth input

Figure 3.2: A Summary of HiRes Operators

`SplitSeq` constructs express the former and latter conditions, respectively, and both divide their inputs into N equal-sized sub-arrays (with the 0th containing any elements in the remainder of that division), and both are merely "shape-changing" operators in that they do not compute anything, nor do they cause, by themselves, any results to be materialized into a new buffer.

Thus, to parallelize `scan1`, we add a new line above line 3:

```
'input := SplitPar('input, THREADS)
```

This changes the meaning of `'input` for the rest of the `Let()`, wherein it now refers to an array-of-vectors type. However, the output of the program also changes to:

$$\bar{\mathsf{A}}\left[\mathsf{V}\left[\mathsf{Rec}\left[(\texttt{key} : \mathsf{Index}, \texttt{value} : \mathsf{Int})\right]\right]\right]$$

This is read as "an array of vectors of records": vectors are always flat containers of data, while arrays are collections of vectors or of other arrays, as Section 3.2 explains more thoroughly.

The num-valid-containing type is likely not what the operator's specification requires, so to fix it we must wrap it in a `Compact(`o`, Offsets(`o`))` operator, which first prefix-sum reduces the valid counts of each vector, and then moves the beginning of each vector to the end of the previous, resulting in a physically-contiguous output, and the elimination of the num-valid type decorator:

```
val op2 = {
  Let(List(
    'input := SplitPar('input, THREADS),
    'output := scan1, // (see prev. listing)
    'offs = Offsets('output)),
  Flatten(Compact('output, 'offs)))
}
```

The choice of when to perform such collection and compaction operations corresponds to the "mask vs. collect vs. compact" dimension of the machine-level plan space described in Chapter 2 (Choice #4).

Partitioning

Radix partitioning is a fundamental building block of efficient joins and aggregations, and a means of inducing parallelism, so HiRes supplies several constructs to support it. First, the `Histogram(`o_{keys}, $slots$`)` operator produces a $slots$-length vector of counts of the number of times each key $k \in \{0, \ldots, slots - 1\}$ appears in o_{keys}. It is assigned a special "built histogram" type $\mathsf{H}_{\text{built}}$ to indicate its intended use in a partition operation, prior to which it must first be *reduced* with `Offsets()`, which yields a new histogram of type H_{off} containing the starting index of each partition's reservation in the underlying buffer. If originally o_{keys} were an array type, then one histogram would be built per array element, yielding type $\mathsf{A}\left[\mathsf{H}_{\text{built}}\right]$.

More code is needed to actually perform a radix partition with, e.g., the 8 LSBs of a 64-bit hash function of some fields:

```
val part: Operator = {
  Let(List(
    'hash := Hash64('input('key), 8),
    'pos := Position('input),
    'key := 'input('key),
    'part :=
      Partition(
        keys = Hash64('input('key), 8),
        values = Project('key, 'pos),
        hist = Offsets(Histogram('hash, 256))),
    'recs := Uncat('part, 0),
    'hist := Uncat('part, 1)),
    RestoreHistogram('recs, 'hist))
}
```

Listing 3.4: Radix Partitioning in HiRes

Here, the `Partition()` primitive actually moves records based on their partition number in the `keys` field, while `Project()` creates a *new flat record* type ($\texttt{key} : \mathsf{Index}, \texttt{pos} : \mathsf{Index}$). The implementation generated for the partition operator uses a *software write-combining buffer (SWWCB)* as proposed by Satish et al. [57] to eliminate cache conflicts and TLB misses. For each partition, the SWWCB allocates one cache line worth of records, and partitioned records are inserted first into their partition's SWWCB slot before being explicitly copied to the appropriate place in the output when that line is full. Since most accesses will be to the SWWCB, which spans a much smaller range of addresses than the full output buffer, fewer TLB entries will be required to address its entries, and fewer TLB misses will be expected per record.

`Hash64()` is a simple multiply-shift hash function [12] cited by Richter et al. [55]. It has been made an `Operator` in its own right, as opposed to a mere expression usable inside of `Eval()`, in order to facilitate future automated tuning transformations by making it easier to "match" joins on the same hash function, though also for the more practical reason that the code it generates in LoRes is more complicated than a single expression, and requires its own block of assignment statements.

The result of partitioning is a pair containing the (still flat) vector of partitioned records, and the histogram, which has been modified in-place as each moved record increments its partition's counter. The final `RestoreHistogram()` construct rotates each histogram's counter array one place to the right to restore the original offsets, and combines it with the flat vector of records to produce an array, which under the hood of the compiler is marked specially to handle its histogram-delineated, variable-length vector elements, but from the programmer's perspective looks just like any other $\mathsf{A}[\mathsf{V}[\ldots]]$ type, yielding a double loop nest for any operator applied to it.

The reason for separating partitioning into so many disjoint primitives is that each one corresponds atomically to a single loop (or loop nest) in the generated code; in practice, these can be grouped together in a HiRes macro, such as `HistRadixPart()` supplied in our standard library. [5]

Since the `Partition()` operator returns *two* outputs–the partition buffer and the histogram–a special `Uncat()` operator is used to extract them separately. It is described with its companion concatenation operator `Cat()` below.

Parallel Partitioning

Although partitioning creates parallelism by forming independently-accessible arrays, its own execution is not inherently parallel. Sadly, split operators alone are no panacea: `Partition()` and `Offsets()` must be made aware of their inputs' nesting to execute in parallel, or else they would simply produce separate, unrelated partitionings of each input sub-array. Three things must be done to merge elements from separate input arrays into a single set of output partitions: first, the `Offsets()` operator must be marked to "merge" independent histograms' counters on a per-partition basis. Second, `Partition()` itself must be annotated with a `parallel` flag to ensure that its output *removes* a layer of nesting from its input, as this will subsequently be added back by `RestoreHistogram()` (effectively, parallel partitioning converts split-based "thread" parallelism into partition parallelism). Inside `Offsets()`, a special `depth` $= n$ argument controls the depth up to which per-partition histogram entries are merged to create per-thread offsets within each output partition buffer. That is, if `depth=1` is set, then array t's nth entry will contain the offset of thread t within the global nth partition. Lastly, since the final, partitioned output should contain merely one histogram counter entry per partition, the original multi-threaded histogram must be reduced with `LastArray()`, which returns the (vector- or array-valued) final element of an array input. In a multi-threaded partitioning, the last thread's offsets (just after `Partition()`-ing) are really the starting offsets of each partition, so they may simply be extracted before use by `RestoreHistogram()`. Listing 3.5 puts this all together.

Loop Fusion Control

Attentive readers may wonder why `Hash64('input('key), 8)` has been repeated twice in the above program, when the `'hash` symbol has already been bound to it. The answer is that the inner-loop fusion of Section 4.3 makes this choice more efficient: since `'hash` is used by the *materializing* (i.e. blocking) operator `Histogram`, on which the subsequent `Partition()` is dependent, use of `'hash` in the latter case would cut *across a loop boundary*, and thus require spilling the entirety of its output to and from memory. In this case, that turns out not to be optimal, although the source needed to recompute `'hash` is a large (32-bit) value in comparison with the 8-bit hash itself, it is *re-read anyways* because it forms part of the

[5] Relatedly, the increase in nesting depth is not applied until `RestoreHistogram()` because this would complicate its integration into the inner-loop fusion algorithm of Chapter 4, which identifies candidate sub-DAGs for fusion based on their depths being equal. Future implementations could fix this with a slight tweak to the algorithm that marks nodes' depth based on their inputs, rather than output

```
val partPar: Operator = {
  Let(List(
    ’input := SplitPar(’input, threads),
    ’hash := Hash64(’input(’key), 8),
    ’pos := Position(’input),
    ’key := ’input(’key),
    ’part :=
      Partition(
        keys = Hash64(’input(’key), 8),
        values = Project(’key, ’pos),
        hist = Offsets(Histogram(’hash, 256), depth=1),
        parallel = true),
    ’recs := Uncat(’part, 0),
    ’hist := Uncat(’part, 1)),
    RestoreHistogram(’recs, LastArray(’hist)))
}
```

Listing 3.5: Parallel Radix Partitioning

partitioned records; in other cases, however, the choice may be less obvious, especially if `’key` were not re-scanned, or if `Hash64` were particularly computationally expensive.

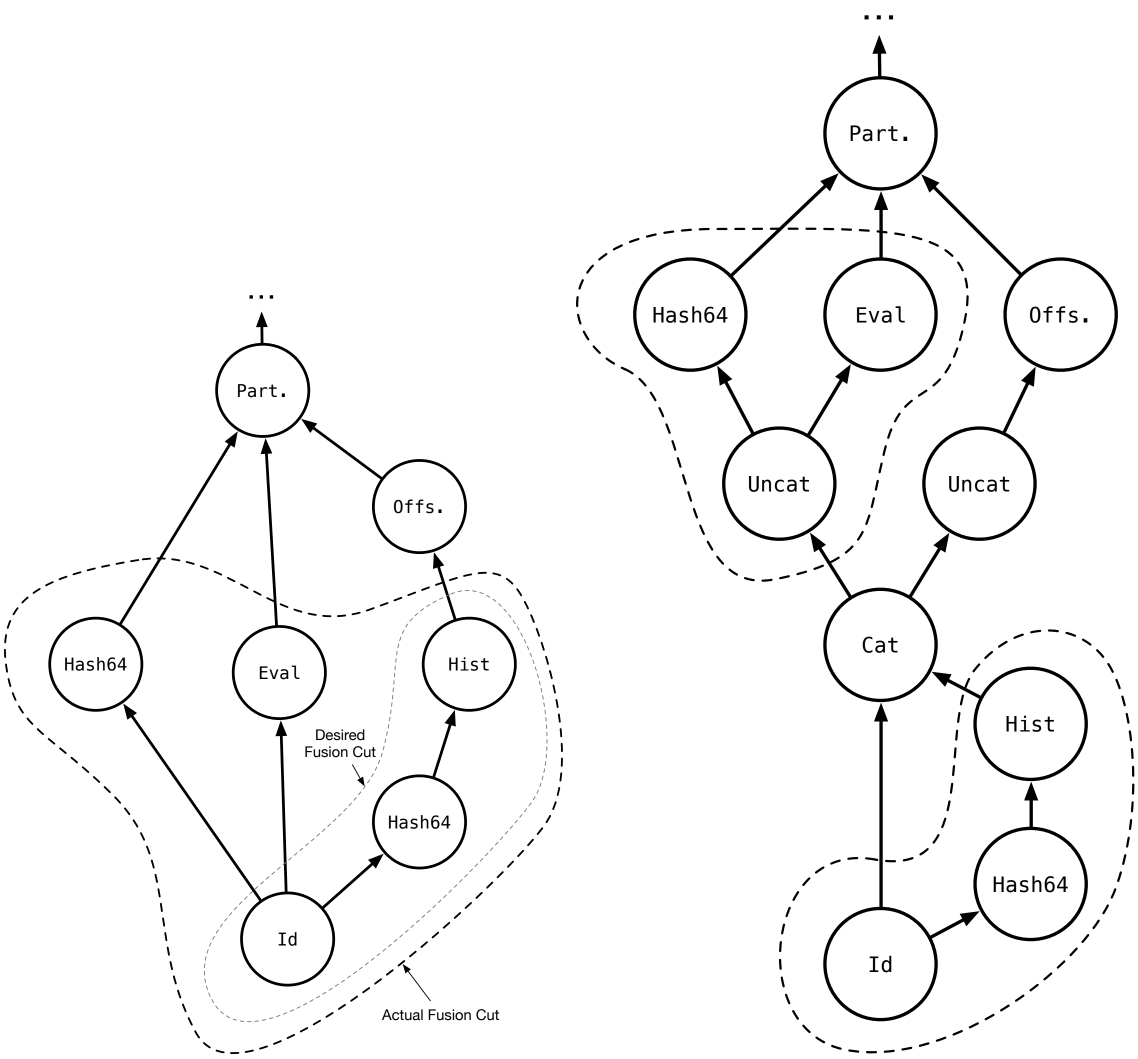
A key contribution of HiRes is allowing this choice to be made, even if the tradeoff may be difficult to analyze. Because the `Let` construct's assignment operator `:=` binds its right-hand side to a *particular buffer*, it allows the programmer to force the re-use of its concrete result. However, were this not desired, it would not actually be necessary to physically re-enter the hashing operator code; instead, *Scala's* assignment operator (`=`) can be used to bind a Scala symbol `hash` to HiRes *AST node*, which would replicate the computation:

```
val hash = Hash64(’input(’key), 8);
```

Fusion Control with `Cat()` and `Uncat()`

Sometimes even the `Let()`-based mechanism is not sufficient to generate the desired loop nest, because nodes may be "pulled" too far forward by the fusion algorithm described in Chapter 4, resulting in an unnecessarily large materialization. Figure 3.3a shows one common example: a `Partition()` operator accepts inputs that can be computed at the same time as the histogram is being built, but since the `Histogram()` operator (along with `Offsets()`) blocks fusion, these additional nodes will be fused with it, forcing their outputs to be materialized across the fusion boundary.

Figure 3.3b, on the other hand, shows how manual insertion of a `Cat()` node, which groups together its inputs as a single operator, forces a different fusion choice by making all subsequent uses of `’input` dependent on the histogram build, thereby preventing them from being fused with it. The concatenation operator returns an output that is unusable on its



(a) Default Fusion for Partitioning (b) Manual Fusion Choice with `Cat()`

Figure 3.3: Manual vs. Automatic Fusion Choices

own. Only by means of an `Uncat(0, N)` can the `N`’th input be recovered: [6]

```
val part: Operator = {
  Let(List(
    ’hash := Hash64(’input(’key), 8),
    ’hist := Histogram(’hash, 256),
    ’cat := Cat(’input, ’hash),
    ’input := Uncat(’cat, 0),
    ’hist := Uncat(’cat, 1),
    ’pos := Position(’input),
    ’key := ’input(’key),
    ’part :=
      Partition(
        keys = Hash64(’input(’key), 8),
        values = Project(’key, ’pos),
        hist = Offsets(hist)),
    ’recs := Uncat(’part, 0),
    ’hist := Uncat(’part, 1)),
    RestoreHistogram(’recs, ’hist))
}
```

Listing 3.6: Better Fusion for Listing 3.4

Hash Tables

Both joins and aggregations are facilitated by a flexible `HashTable` operator, shown here from an aggregation in TPC-H Query 17:

```
val htbl = {
  HashTable(
    ’lineitem.project(
      ’l_partkey -> ’l_partkey,
      ’l_quantity -> ’l_quantity,
      ’count -> DoubleConst(1.0)),
    aggregates = List(
      (NFieldName(’l_quantity), PlusOp),
      (NFieldName(’count), PlusOp)),
    hash =
      Some(Hash64(’lineitem(’l_partkey), nbits)),
    buckets = Some(1 << nbits))
}
```

The `HashTable` operator uses its input’s record format as the format of the hash table, and accepts a user-specified hash function whose range must be less than or equal to the supplied number of `buckets`. In building the hash table, a probe is assumed to match iff *all* fields of the input record match their corresponding fields of the test record, except those

[6] This grouping mechanism is also used internally by the compiler to implement the fusion pass, so the “default” fusion choice will insert `Cat()`s to return multiple outputs as well.

fields specifically marked by name as *aggregates*, like `'l_quantity` and `'count` are above, in which cases the indicated aggregation operation is performed. An optional `slots` argument sets the size of each bucket's initial allocation. Hash tables are built using a special *chunk array* data structure (Figure 3.7) whose vector elements expand dynamically once an initially reserved allocation is exceeded. The result type adds one layer of chunk array nesting (A^C []) to the input type (clearing any others), to indicate that its elements are stored in physically separate linked list nodes. Chunk nesting can only be removed via `Compact()` (Section 3.1).

Joins

The `HashJoin(left, hash, right)` operator, by contrast, expects a nested array-typed value as its right input, which is probed with records from the left. A hash function–supplied as a third input–simply selects which element of (the innermost dimension of) the right-side array to loop over. Thus, a simple nested loops join can be constructed with a one-element right-side array. A boolean-valued `test` expression–with access to all fields of an output record formed by appending all the right-side fields to those of the left–decides if a probe is a match.

The right hand side must be an array, but it need not be the chunked array output of a `HashTable`. Thus, it is possible to construct a hash table multiple ways: one can can `Partition()` to the appropriate number of buckets, or use the `Compact()`'ed output of a previously-built table, which may be superior for non-uniform key distributions.

By default, the output of a join is itself a chunked-array of length one, which expands by allocating more chunks if the size of the join output exceeds its predicted allocation. If the left-side input is an array, however, then the join array will take on the left (probe) array's length. In this way, parallel radix joins are processed partition-by-partition, with outputs maintaining the structure of the probe input. It also means that `SplitSeq(o,` N`)` can be used to to construct an *interleaved join* (i.e. one that leverages the temporal parallelism of Figure 2.4b to achieve the "block-wise" execution of Vectorwise [76]), in which two consecutive joins are executed for one block of the first join's probe input, yielding a materialized output, which is then fed as the probe input to the second join, as the materialization only interrupts inner-loop fusion, but not outer-loop fusion[7].

Indexed Joins & Gather

When no attribute packing (machine-level plan dimension #3 of Chapter 2) is used, hash tables store only a key and a row-ID (*rid*), which permits successful probes to re-assemble other attributes from the original relation by gathering them from their respective column arrays. HiRes supplies a `Position()` operator that returns the either the relative or absolute index (with scalar type `Index`) of every element in its input, and which can be projected (or zipped) into a hash table build. Subsequently, a binary `Gather(`*src*`,` *dst*`)` operator retrieves

[7] Ideally we would like to interleave without any materialization, but this would require substantial revision of the current fusion architecture, and so is reserved for future work.

elements located at the Index-typed positions in *src* from *dst*, which generally should be a flat vector if absolute indices are used. Only if both sides of the join were already parallelized before the build (e.g. by partitioning) should relative indices be used, in which case the build (right) side would be nested.

Reductions

SQL `sum()`, `avg()`, `count()`, and other functions cause *reductions* of relational attributes. Depending on context, there are several different ways to realize their semantics in HiRes, which supplies multiple primitives for implementing generic reduction operators.

Flat (Vector) Reductions

The simplest reducer is `Reduce()`:

```
val avg: Operator = { // Assume ‘’input‘ has type ‘V[UInt32]‘
  Let(
    List(
      ’sum := Reduce(’input, PlusOp),
      ’count := Reduce(’input(Const(1)), PlusOp)),
    in = Zip(’sum, ’count)(’sum / ’count))
}
```

Listing 3.7: A `avg()` Implementation in HiRes

Listing 3.7 demonstrates the construction of an average operator using the `Reduce()` primitive, which takes an input vector and an aggregation operation (here `+`) to produce a scalar-typed result. Recall that `Zip()` introduces the `'sum` and `'count` symbols into a single record namespace, allowing the implicit `Eval()` on the last line to access them. Additionally, the `'input(Const(1))` expression yields a vector-typed value whose elements are all `1` and whose length is the same as that of `'input`. If `'input` were masked, this vector would be masked too, resulting in a count of only those elements with a set valid bit in the predicate mask vector.

There is no primitive average operator because such a construct would entail allocating a separate counter and performing a post-reduction division as in the above example, thus violating the single-loop-per-primitive principle. However, the pattern of Listing 3.7 can easily be templated into a macro, such as the HiRes Standard Library’s `SumDouble()` operator.

Nested (Array) Reductions

Vectors can be reduced to scalars with `Reduce()`, but *arrays of scalars* (see Section 3.2) cannot. The `NestedReduce()` primitive serves this purpose, and helps to parallelize reductions. It removes one layer of array nesting, and if a vector type is present, then the nth element of each vector is reduced with the nth scalar element of each other vector. Thus, when a sub-query containing reductions is parallelized with `SplitPar()`, it must be completed with a final `NestedReduce()` much in the same way as an ordinary parallelization would require a `Flatten()` at the end:

```
val avgPar: Operator = { // Assume ‘’input‘ has type ‘V[UInt32]‘
  Let(
    List(
      'input := SplitPar('input),
      'sum := Reduce('input, PlusOp),
      'count := Reduce('input(Const(1)), PlusOp),
      'sum := NestedReduce('sum, PlusOp),
      'count := NestedReduce('count, PlusOp)),
    in = Zip('sum, 'count)('sum / 'count))
}
```

Listing 3.8: Parallelizing Listing 3.7

Sorting

Sorting has not yet been a primary bottleneck in queries optimized with HiRes, so it does not yet offer a full panoply of sorting operations. At this time, the only one supplied natively is insertion sort, though surely future versions of the language would require a wider range of more efficient building blocks, such as a merge operator (for mergesort), a hardwired quicksort primitive, and other more data-parallel algorithms as well. However, reasonably efficient sorting routines can still be constructed by combining insertion sort with partitioning, wherein it acts as the final stage of (possibly multi-pass) radix sort.

The `InsertionSort(`$o, k_1 \ldots k_n$`, )` construct is materializing and, like other data movement operators, reproduces the input record format in its output. Here, $k_1 \ldots k_n$ are the names (or indices) of number-valued key fields within the input record used to determine the (ascending) sort order; wherever two records have the same value at key $k_1 \ldots k_i$, then k_{i+1} will be used to break the tie. In contrast to partitioning, the keys must necessarily reside in the output, in order for such comparison to take place, but since the input need not be material, zip and projection adjustments within a fused operator may reshape it before sorting materializes it.

3.2 HiRes Type System

The type system of HiRes is designed to facilitate both memory layout and execution flow optimizations. It is presented in a symbolic representation, summarized in Figure 3.4, for the sake of brevity and clarity.

Vector and Array Types

The fundamental unit of data in HiRes is a *vector*, or an extent of data with contiguous indices, denoted by $\mathsf{V}[S]$ for scalar elements of type S. A vector corresponds to an

P	::=	Index, Bool, Float, Double	Primitives
		Int8, Int16, Int32, Int64	
		UInt8, UInt16, UInt32, UInt64	
Rec	::=	$\mathsf{Rec}\,[\mathsf{f}_1, \ldots, \mathsf{f}_n]$	Flat Records
		$\mathsf{Rec}\,[(\mathsf{f}_1, \ldots, \mathsf{f}_n), \ldots, (\mathsf{f}_1, \ldots, \mathsf{f}_m)]$	Split Records
f	::=	$([name] \ : \ \mathsf{P})$	Record Field
$\mathbf{S}$	::=	$\mathsf{P}, \ \mathsf{Rec}$	Scalars
$\mathbf{V}$	::=	$\mathsf{V}\,[\mathbf{S}]$	Vectors
		$\bar{\mathsf{V}}\,[\mathbf{S}]$	Vectors with num-valid counter
		$\mathsf{V}^{P}\,[\mathbf{S}]$	Vectors with predicate mask
$\mathbf{A}$	::=	$\mathsf{A}\,[\mathbf{T}, (\mathbf{H})]$	Arrays (with any decorators)
		$(\mathsf{A})\,[\mathbf{T}]$	Arrays with *no* decorators
		$\bar{\mathsf{A}}\,[\mathbf{T}]$	Arrays with num-valid counters
		$\mathsf{A}^{P}\,[\mathbf{T}]$	Arrays with padding
		$\mathsf{A}^{C}\,[\mathbf{T}]$	Chunked arrays
		$\mathsf{A}^{D}\,[\mathbf{T}]$	Disjoint Arrays
$\mathbf{H}$	::=		Histograms
		$\mathsf{H}_{\text{built}}$	Built histogram (incremented)
		H_{off}	Reduced histogram (prefix sum)
		H_{end}	Histogram after moving records
$\mathbf{T}$	::=	$\mathbf{S}, \mathbf{V}, \mathbf{A}, \mathbf{H}$	All types

Figure 3.4: HiRes Types

inner loop of a record-processing operator. *Nesting* operators–such as `SplitPar()` and `RestoreHistogram()`–turn a vector into an *array of vectors* $\mathsf{A}\,[\mathsf{V}\,[S]]$, or even arrays of arrays of vectors $\mathsf{A}\,[\mathsf{A}\,[\mathsf{V}\,[S]]]$, where each degree of array nesting means one more degree of *outer loop* nesting. Since there can be at most one inner loop, at most one degree of vector typing is allowed. The vector and array levels align roughly with data-level and thread-level parallelism, respectively.

Not only arrays of vectors, but also arrays of scalars $\mathsf{A}\,[S]$ are possible, and indeed meaningful: if a binary operator's inputs have types $\mathsf{A}\,[\mathsf{V}\,[S_1]]$ and $\mathsf{A}\,[S_2]$, then each of the latter arrays' scalar elements would be *broadcast* (repeated) across all scalar elements in the corresponding array of the former. Conversely, with inputs typed $\mathsf{V}\,[S_1]$ and $\mathsf{A}\,[\mathsf{V}\,[S_2]]$, the *vectors* of the former would be broadcast across the arrays of the latter. Figure 3.6 illustrates

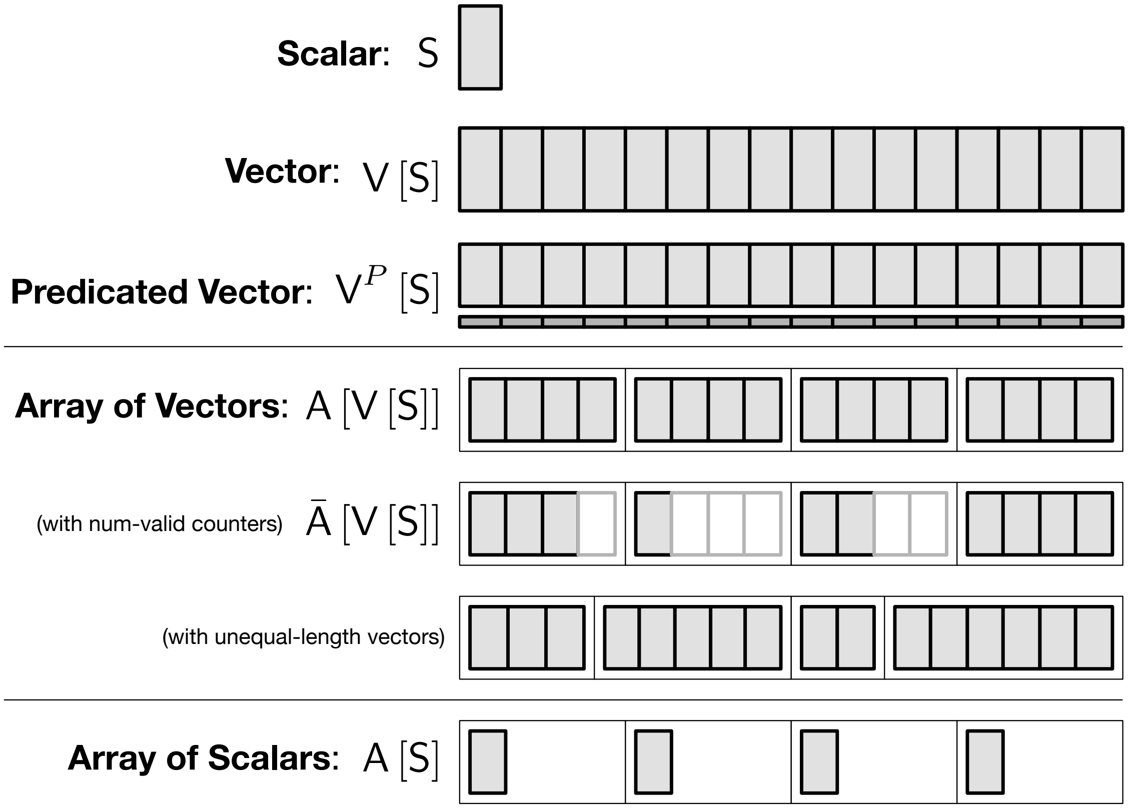

Scalar: S
Vector: V [S]
Predicated Vector: V^P [S]
Array of Vectors: A [V [S]]
(with num-valid counters) Ā [V [S]]
(with unequal-length vectors)
Array of Scalars: A [S]

Figure 3.5: HiRes Array and Vector Types

both of these properties of array and vector types.

The HiRes type system also aims to ensure that only valid records are used by and returned from HiRes programs. To that end, array and vector types may also have *decorators*, which indicate the presence of metadata such as *masks*–bit-vectors marking which records are active and indicated with the superscript $\mathsf{V}^P\,[S]$. When a vector is masked, operations on its elements will be wrapped in `if` statements (in the lower-level, C-like intermediate LoRes) using the mask as a condition to avoid computation and memory accesses for invalid rows. A decorator also exists for *num-valid* counters, which are marked $\bar{\mathsf{A}}\,[\mathsf{V}\,[S]]$ for an array whose vectors each contain valid data in only an initial subset of their physical space. Fixed numbers of dummy *padding* elements at the end of each (vector) element of $\mathsf{A}^P\,[\mathsf{V}\,[S]]$ may be marked as well. *Chunked* decorators $\mathsf{A}^C\,[\mathsf{V}\,[S]]$ and *disjoint* decorators $\mathsf{A}^D\,[\mathsf{V}\,[S]]$ indicate that some portion of each vector is stored in physically disjoint memory (see the next section). In order to avoid the use of rows with non-valid data, the HiRes type checker rejects programs that attempt to discard such metadata, insisting that any flattening operation either perform *compaction* (Section 3.1), or that the array type in question be *pure* (indicated by parentheses: $(\mathsf{A})\,[\ldots]$), and contain no decorator.

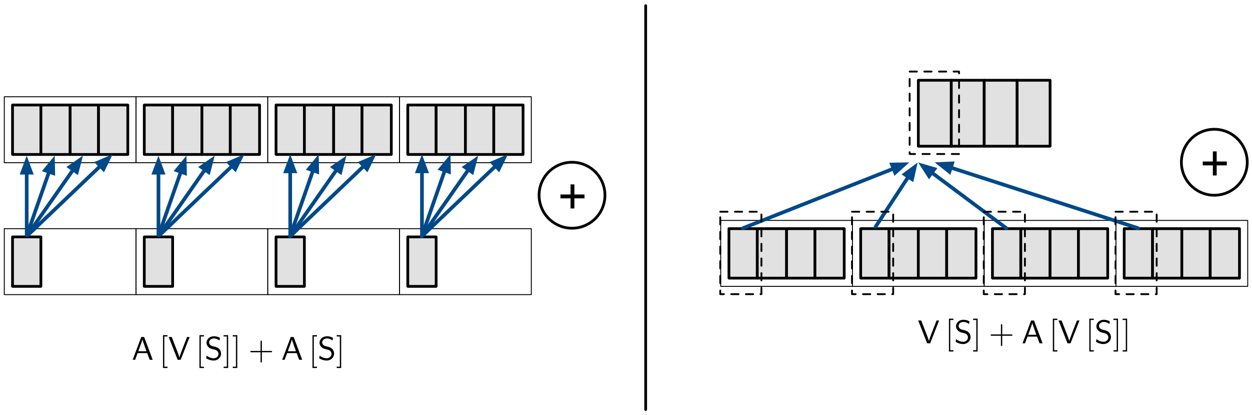


Figure 3.6: Depth Promotion: Unequal depth results in scalar or vector broadcast

Chunked Arrays & Hash Tables

Most HiRes datatypes assume to be backed by fixed-length, physically contiguous buffers (at least, one for each field group), but hash tables and join results may require unpredictable amounts of per-bucket storage. For this reason, HiRes includes an efficient "chunked hash table" data structure, shown in Figure 3.7, that optimizes for the common case of (hopefully) well-chosen overload factors, permitting each bucket's initial allocation to reside in a physically contiguous base buffer, while also enabling graceful overflow to linked lists of bucket-sized *chunks* of supplemental storage. It is a kind of compromise between chaining and linear probing in that probes to a given bucket will scan, in sequence, the all records in a chunk before traversing the "chain" of further chunks, if any exists. In generating code to traverse a chunked array, the compiler adds one extra loop level to any iteration over its "vector" elements to traverse the chain linked list nodes, but this is not semantically exposed (it does not affect broadcast properties, for example), and in the common case should have a bound of one.

A chunked array type $\mathsf{A}^C\,[\mathsf{V}\,[S]]$ (of which any type may contain at most one chunk decorator) thus closely resembles a normal array type $\bar{\mathsf{A}}\,[\mathsf{V}\,[S]]$ with N (vector) elements in that its underlying buffer contains space for N fixed-length vectors, and num-valid counters in a separate array, but with further ancillary data structures to manage the bucket's chunk lists. However, storing counters separately would result in very inefficient probes: if the first probe to a bucket of a large table is expected to cause a cache miss, then so too would the access to the separately stored counter indicating how many slots are occupied. For this reason, an *inline counter* mode is supported, that allocates an extra "dummy" slot just before the start of each bucket, in which the bucket's counter is stored[8]. If this counter is greater than the chunk size, then the ancillary list arrays will be consulted, but not otherwise.

[8] Our current prototype requires that the first field of the table's record type be large enough to count the number of records in a chunk; thus if it is a `UInt8` type, the chunk size must be no larger than 256 records.

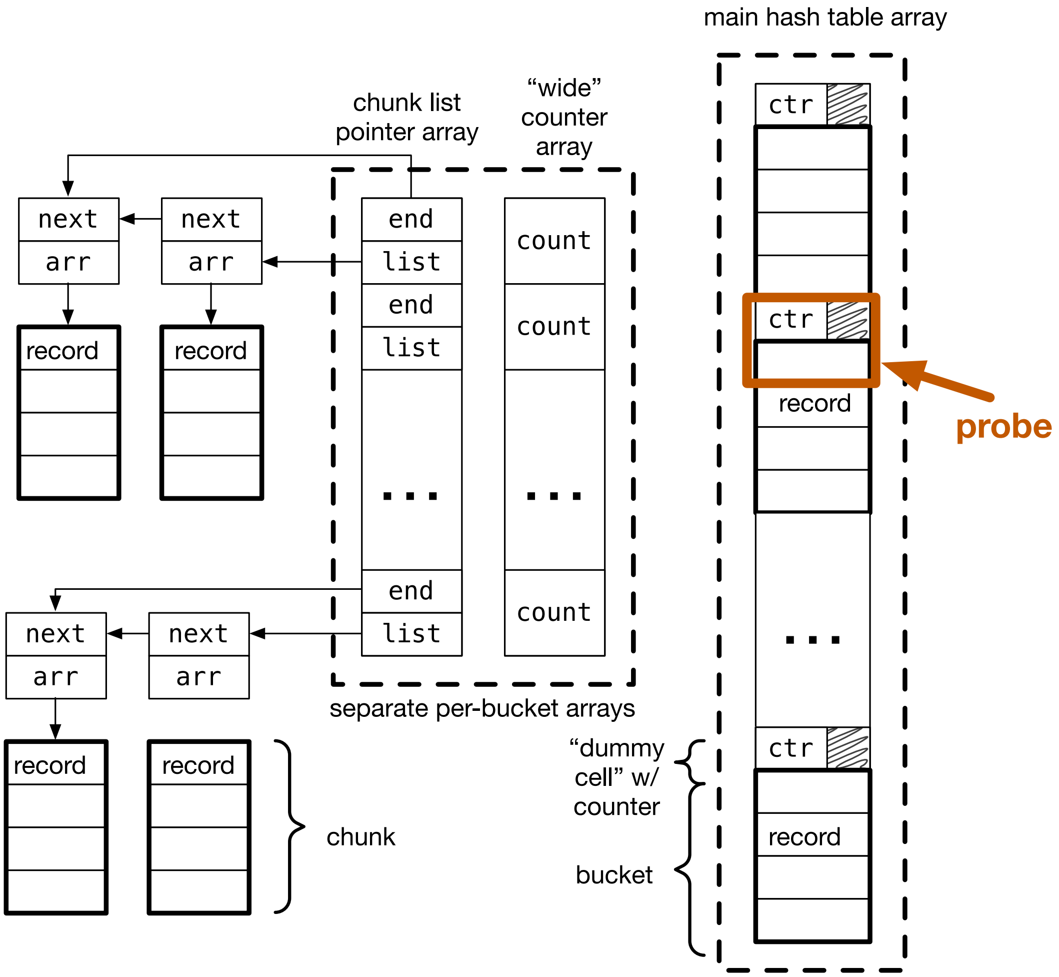


Figure 3.7: Efficient "Chunked" Hash Table Format

Because this chunked array type supports the same split and flat record types as regular arrays, it is possible to create buckets with contiguous extents of a single field (as its own field group) which, if `Split()` appropriately, would be amenable to SIMD processing in future versions of the compiler that support it. Alternatively, when packing (row-major order) is beneficial, this is supported too.

Disjoint Arrays & Ancillaries

Figure 3.8 depicts another possible array structure in which each element (vector or array) resides in a *disjoint* allocation of memory. Unlike chunked arrays, disjoint arrays contain only one pointer per extent, rather than a linked list of chunks, though the extents may have different lengths if there is a histogram attached to track them. They can have num-valid and predicate mask decorators as well.

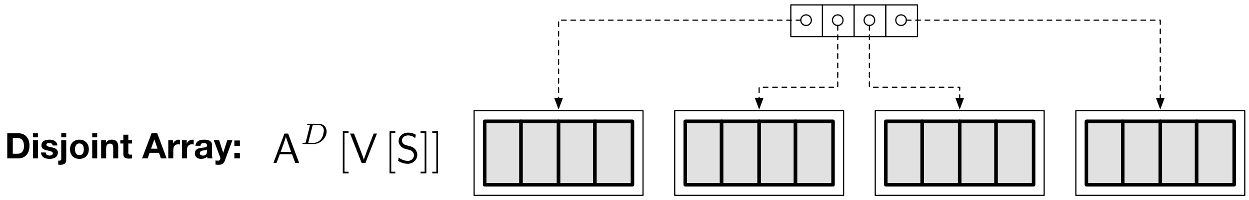


Figure 3.8: Disjoint Arrays

Disjoint arrays are an important performance optimization in cases where parallel accesses to contiguous sub-arrays would otherwise cause false sharing or incur the cost of crossing NUMA domains. Thus, when operators like `Histogram()` produce *ancillary* data structures from nested inputs, they default to disjoint-typed outputs, and split-like operators can be annotated with a `disjoint` directive to induce this layout manually. The type-checking rules in Section 3.2 ensure, as above, that disjoint arrays cannot be flattened without compaction, or otherwise used where contiguous buffers are required. To ensure the maximum benefit from chunked arrays, when the compiler generates allocation code for them, it marks the allocation loop over all sub-arrays as parallel, so that each one will be allocated by the thread that ultimately uses it[9], thereby ensuring that threads will use memory within their own NUMA domains.

Record & Relation Structure

While array-vector nesting controls execution by imposing loop structure, the concrete type of scalar S itself determines memory *layout*. In addition to simple primitive types such as integers and booleans, scalars in HiRes can also be structured record types: a *flat record* contains contiguous *fields* packed together in a struct, while a *split record* has multiple *groups* of fields that each reside in separate physical columns of memory when that record type is instantiated in a vector. Thus, the HiRes type system partially orthogonalizes execution concerns–how rows are divided between loop iterations and which ones are valid–from storage concerns and how memory for each relation and intermediate result is accessed.

Records consist of ordered fields of primitive types and optional names. A flat record is denoted by

$$(\texttt{key} : \mathsf{UInt32}, \texttt{value} : \mathsf{Index})$$

for a tuple with a 32-bit unsigned `key` field, and a `value` field equal in size to the native C `size_t` type.

The primitive types currently supported by HiRes include $\{\texttt{UInt}, \texttt{SInt}\} \times \{8, 16, 32, 64\}$, as well as Bool, Int, Index, and single- and double-precision floating-point types. Meanwhile, a split record denoted by $((\mathsf{Index}), (\mathsf{UInt8}, \mathsf{UInt8}))$ contains, in this case, one column of Index-

[9] Assuming the `OMP_PROC_BIND` environment variable is appropriately set to associate each hardware thread with a consistent loop index

typed values, and another with pairs of 8-bit unsigned values. HiRes does not currently support non-byte, non-power-of-two sized datatypes, and does not handle strings.

Typing Rules

HiRes is *implicitly typed*, meaning that the type of an operator is inferred by the compiler without any explicit annotation by the programmer, aside from the initial marking of input and output relation types; inside an operator, new record types are created by manipulation with `Zip`, `Project`, and `Let`, while array and vector types are manipulated with nesting and flattening operators, and implicitly by combination of types with differing depths. The rules that govern these inferences assume each concrete operator included in the syntax of Figure 3.2 has a *type signature* specifying its *canonical function type* in terms of the minimal nesting depths expected of its inputs. When more complex (i.e. deeper) inputs are provided, the rules of this section determine how to adapt those canonical rules.

Nesting and Flattening

Figure 3.12 explains the rules for nesting and flattening. The `Split` operators straightforwardly add an array layer $\mathsf{A}\,[T]$ to an input type T, where the latter is a *type variable* that stands for any type, so long as it satisfies the rule's premise (in this case, it must not be a scalar); when followed by square brackets, $T_1[T_2]$ indicates that T_1 refers to some layers of nesting on top of type T_2. The Flat rule restricts the use of flattening operations to those cases where no decorators are present at the level being flattened, as discarding them would lead to the use of invalid elements.

Rules prefixed NV- govern the introduction and elimination of num-valid counters, which are stored out-of-band with the array's data proper. These counters are introduced by `Collect`, which does not alter the size or position of any array, but does render the tail of each vector invalid. Figure 3.10 also stipulates how the adjustments of nesting and flattening changes propagate across subsequent operators that use them.

In the case of a single-input operator, the ArrUp rule simply indicates that a vector-vector operation of signature $\mathsf{V}\,[\mathsf{S}_1] \to \mathsf{V}\,[\mathsf{S}_2]$ becomes an array-array operation in which the output takes on the same depth (and structure) as the input. Its first premise should be read as "given an N-input operator o with type signature $(\mathsf{V}_1, \ldots, \mathsf{V}_n)$...". When its concrete inputs $o_1, \ldots, o_N$ have different structures, however, the other two premises select a type T_M having the *greatest depth* of all, and yield a result type of equal depth. In these cases, the *substitution operator* (forward slash) replaces the innermost vector type while keeping the rest of the structure intact; similar rules (Subst and SubstIn) apply when the right hand side is nested, replacing up to the nesting depth of that side.

Depth Promotion & Broadcast

The auxiliary rules in Figure 3.10 serve two functions: they define the *nesting depth equivalence* relation ≡, and ensure that chunk decorators occur at the same depth of each input in a multi-input operator. The latter condition is necessary for correct loop-nest generation: if it were not enforced, it would not be possible to add a single loop over all chunk lists, and, more importantly, it would likely indicate that the inputs' chunks were generated from different sources with different lengths, which would result in incorrect operation.[10] The chunk alignment condition is enforced by splitting the definition of depth equivalence between the EqUp and EqUpChunk rules: an equivalent pair of nested types can be made from a pair of equivalent base types only if either both sides add chunking, or both do not. The condition $T_M \geq T_{i \leq N}$ in ArrUp ensures that the deepest input is equivalent to all other inputs up to their depths, which guarantees in turn that any chunks present in one input are present in all inputs at the same level. Note that the typing rules for chunk-producing operators (`HashTable`) ensure that at most one layer of chunking exists for any result.

The type signature of `HashJoin` does not conform to ArrUp because of its second input's nested (array) type, so separate rules in Figure 3.11 extend its domain of well-typed inputs to higher depths. The rules for `HashJoin` are even more complex than its signature implies, since it actually only adds nesting when the first input has none. Despite its arbitrary appearance, this rule is motivated by the desire to allow chained joins, as described in Section 3.1: if the probe is a flat vector, then a length-one chunked array is added onto the output to contain its unknown and arbitrary number of match elements, but if the probe input is already nested, this depth is simply preserved in the output, which is now chunked at the innermost array level, meaning that each array of the input produces its own run of match elements in the output (see Section 3.1).

Disjoint Array Typing

The typing rules for disjoint arrays in Figure 3.13 ensure not only that non-contiguous data cannot be "flattened", but also that the specific structure of discontinuity expressed is coherent. For example, an already-nested array subjected to a disjoint `Split()` must return a result that is disjoint at *all* levels (Rule DisjA)–rather than just the innermost, newest level–as the physical separation of innermost vectors necessarily implies that outer levels, which group together multiple vectors each, are physically disjoint too. The property of disjointedness thus differs from from chunkiness in that it can be marked on multiple levels, though only so long as it extends to all outermost nesting depths beyond the first at which it is marked.

Disjointedness does not alter nesting equivalence (rules EqUp and EqUpChunk), because multi-input operators can process sources with disjointedness on different levels without any difficulty, as there is no chunk loop to coordinate. Most operators do not even need to

[10] The type system cannot fully guard against this possibility; instead, it can be checked during the generation of intermediate data structures that represent the lengths of each array dimension explicitly

$$\frac{T_2 \equiv T_3}{T_1[T_2]/T_3 = T_1[T_3]}\ \textsc{Subst} \qquad \frac{T_2 \geq T_3 \wedge \neg\,(T_2 \equiv T_3)}{T_1[T_2]/T_3 = T_1[T_2/T_3]}\ \textsc{SubstIn}$$

$$\mathsf{S}_1/\mathsf{S}_2 = \mathsf{S}_2 \qquad \mathsf{V}_1[T_1]/T_2 = \mathsf{V}_1[T_1/T_2] \qquad \mathsf{A}_1[T_1]/T_2 = \mathsf{A}_1[T_1/T_2]$$

Figure 3.9: Type Substitution Rules

$$\mathsf{S}_1 \equiv \mathsf{S}_2\ \textsc{ScalEq} \qquad \mathsf{V}_1 \equiv \mathsf{V}_2\ \textsc{VectEq}$$

$$\frac{T_1 \equiv T_2}{\mathsf{A}^{\bar{C}}\,[T_1] \equiv \mathsf{A}^{\bar{C}}\,[T_2]}\ \textsc{EqUp} \qquad \frac{T_1 \equiv T_2}{\mathsf{A}^{C}\,[T_1] \equiv \mathsf{A}^{C}\,[T_2]}\ \textsc{EqUpChunk}$$

$$\frac{T_1 \equiv T_2}{\neg\left(\mathsf{A}^{C}\,[T_1] \equiv \mathsf{A}^{\bar{C}}\,[T_2]\right)}\ \textsc{NonEq} \quad \frac{T_1 \equiv T_2}{T_1 \geq T_2}\ \textsc{GtEq} \quad \frac{T_1 \geq T_2}{\mathsf{A}\,[T_1] \geq T_2}\ \textsc{GtEqUp}$$

$$\frac{o\ :\ (\mathsf{V}_1, \ldots, \mathsf{V}_N) \rightarrow \mathsf{V}_R \qquad o_{i \leq N}\ :\ T_i \qquad T_M \geq T_{i<N}}{o\,(o_1, \ldots, o_N)\ :\ (T_1, \ldots, T_n) \rightarrow T_M/\mathsf{V}_R}\ \textsc{ArrUp}$$

Figure 3.10: Array Broadcast (Depth Promotion) Rules

be aware of disjointedness, with the exception of `Split()` and `Partition()`, which generate it, `Offsets()`, which must clear its counter accumulation at every discontinuity (so that each separate partitioning instance starts at index zero relative to its disjoint buffer), and ancillary-creating operators like `Histogram()`.

The propagation of disjoint decorators upwards to all nesting levels (DisjA) clears any chunk and num-valid decorators, since whatever information these might have carried is now encoded in the innermost $\mathsf{A}^D\,[T]$ type. Under the hood, the concrete data structure generated for the disjoint type will contain a set of num-valid counters, but this need not be reflected in the type itself, as $\mathsf{A}^D\,[T]$ already implies that a compaction is necessary before flattening is permissible. It is perfectly sensible to apply further, non-disjoint `Split()`s, resulting in a mixed type such as $\mathsf{A}^D\,[\mathsf{A}\,[\mathsf{S}_1]]$. This mixed nesting enables independent processing of contiguous extents within arrays that are themselves disjoint, and it can easily be removed with a simple `Flatten()`.

$$\frac{o_1 \;:\; T_1 \qquad o_2 \;:\; T_2 \qquad T_1 \equiv T_2 \qquad T_3 \geq \mathsf{A}^C\left[\mathsf{V}\left[\mathsf{S}_3\right]\right]}{\texttt{HashJoin}(o_1, o_2) \;:\; T_3}\ \textsc{Join}$$

$$\frac{o_1 \;:\; \mathsf{V}_1 \qquad o_2 \;:\; \mathsf{A}\left[\mathsf{V}_2\right] \qquad T_3 \geq \mathsf{A}^C\left[\mathsf{V}_3\right]}{\texttt{HashJoin}(o_1, o_2) \;:\; \mathsf{A}^C\left[\mathsf{V}_3\right]}\ \textsc{JoinUp}$$

Figure 3.11: Hash Join Typing Rules

$$\frac{O \;:\; T_1 \qquad T_1 = T_2[\mathsf{V}_3]}{\texttt{Split(}O\texttt{)} \;:\; T_2\left[\mathsf{A}\left[\mathsf{V}_3\right]\right]}\ \textsc{Split} \qquad\qquad \frac{O \;:\; T_1[(\mathsf{A})\left[\mathsf{V}_1\right]]}{\texttt{Flatten(}O\texttt{)} \;:\; T_1[\mathsf{V}_1]}\ \textsc{Flat}$$

$$\frac{O \;:\; T_1[T_2^P]}{\texttt{Collect(}O\texttt{)} \;:\; \bar{T}_1[T_2]}\ \textsc{NVIntro} \qquad \frac{O \;:\; \bar{T}_1[T_2^P]}{\texttt{Collect(}O\texttt{)} \;:\; \bar{T}_1[T_2]}\ \textsc{NVProp}$$

$$\frac{O_1 \;:\; T_1\left[\bar{\mathsf{A}}\left[\mathsf{V}_1\right]\right] \qquad O_2 \;:\; T_1[\mathsf{H}_{\text{off}}]}{\texttt{Compact(}O_1\texttt{, hist=}O_2\texttt{)} \;:\; T_1\left[(\mathsf{A})\left[\mathsf{V}_1, \mathsf{H}_{\text{end}}\right]\right]}\ \textsc{NVDrop}$$

Figure 3.12: Nesting-Flattening Rules

$$\frac{T_1 = \mathsf{A}\left[T_1\right]}{\mathsf{disj}\left(T_1\right) = \mathsf{A}^D\left[\mathsf{disj}\left(T_2\right)\right]}\ \textsc{DisjA} \qquad \frac{T_1 = \mathsf{V}_1}{\mathsf{disj}\left(T_1\right) = T_1} \qquad\qquad \frac{T_1 = \mathsf{S}_1}{\mathsf{disj}\left(T_1\right) = T_1}$$

$$\frac{O \;:\; T_1 \qquad T_1 = T_2[\mathsf{V}_3]}{\texttt{Split(}O\texttt{,disjoint)} \;:\; \mathsf{disj}\left(T_2\left[\mathsf{A}^D\left[\mathsf{V}_3\right]\right]\right)}\ \textsc{SplitD}$$

Figure 3.13: Disjoint Array Typing Rules

$$\frac{\Gamma \vdash o_1 \;:\; T_1 \qquad \Gamma, x_1 \;:\; T_1 \vdash o_2 \;:\; T_2}{\Gamma \vdash \texttt{Let(}[x_1 = o_1]\texttt{, in } o_2\texttt{)} \;:\; T_2}\ \textsc{LetBase}$$

$$\frac{\Gamma \vdash x_1 \;:\; T_1 \qquad \Gamma, x_1 \;:\; T_1 \vdash \texttt{Let(}[\ldots, x_n = o_n]\texttt{, in } o_m\texttt{)} \;:\; T_m}{\Gamma \vdash \texttt{Let(}[x_1 = o_1, \ldots, x_n = o_n]\texttt{, in } o_m\texttt{)} \;:\; T_m}\ \textsc{LetProg}$$

Figure 3.14: Let Typing Rules

$$\frac{T = \mathsf{V}\left[\mathsf{P}\right]}{\mathrm{groups}\left(T\right) = \left[\left(\mathsf{P}\right)\right]} \qquad \frac{T = \mathsf{V}\left[\mathsf{Rec}\left[\left[f_1, \ldots, f_n\right]\right]\right]}{\mathrm{groups}\left(T\right) = \left[\left(f_1, \ldots, f_n\right)\right]} \qquad \frac{T = \mathsf{V}\left[\mathsf{Rec}\left[\left[g_1, \ldots, g_n\right]\right]\right]}{\mathrm{groups}\left(T\right) = \left[g_1, \ldots, g_n\right]}$$

$$\frac{\begin{array}{c} o_{i \leq n} \ : \ \mathsf{V}\left[\mathsf{P}_i\right] \\ T = \mathsf{V}\left[\mathsf{Rec}\left[\left(\mathtt{f}_1 \ : \ \mathsf{P}_1\right), \ldots, \left(\mathtt{f}_n \ : \ \mathsf{P}_n\right)\right]\right] \\ O = \mathtt{Project(}\mathtt{f}_1, \ldots, \mathtt{f}_n\mathtt{)} \end{array}}{\mathtt{Let(}\left[\mathtt{f}_1 = o_1, \ldots, \mathtt{f}_n = o_n\right], \ \mathtt{in} \ O\mathtt{)} \ : \ T} \ \textsc{LetProjS}$$

$$\frac{\begin{array}{c} o_{i \leq n} \ : \ \mathsf{V}\left[\mathsf{S}_i\right] \qquad f_{ij} = \left(\mathrm{groups}\left(o_i\right)\right)_j \forall_{j \leq |\mathrm{groups}(o_i)|} \forall_{i \leq n} = \left(\mathtt{f}_j \ : \ \mathsf{P}_j\right) \\ T = \mathsf{V}\left[\mathsf{Rec}\left[f_{11}, \ldots, f_{nm}\right]\right] \qquad O = \mathtt{Project(}\mathtt{f}_1, \ldots, \mathtt{f}_n\mathtt{)} \end{array}}{\mathtt{Let(}\left[\mathtt{f}_1 = o_1, \ldots, \mathtt{f}_n = o_n\right], \ \mathtt{in} \ O\mathtt{)} \ : \ T} \ \textsc{LetProj}$$

$$\frac{\begin{array}{c} o_{i \leq n} \ : \ \mathsf{V}\left[\mathsf{P}_i\right] \\ T = \mathsf{V}\left[\mathsf{Rec}\left[\left(\left(\mathtt{f}_1 \ : \ \mathsf{P}_1\right)\right), \ldots, \left(\left(\mathtt{f}_n \ : \ \mathsf{P}_n\right)\right)\right]\right] \\ O = \mathtt{Zip(}\mathtt{f}_1, \ldots, \mathtt{f}_n\mathtt{)} \end{array}}{\mathtt{Let(}\left[\mathtt{f}_1 = o_1, \ldots, \mathtt{f}_n = o_n\right], \ \mathtt{in} \ O\mathtt{)} \ : \ T} \ \textsc{LetZipS}$$

$$\frac{\begin{array}{c} o_{i \leq n} \ : \ \mathsf{V}\left[\mathsf{S}_i\right] \qquad g_{ij} = \mathrm{groups}\left(\mathsf{S}_i\right)_j \\ T = \mathsf{V}\left[\mathsf{Rec}\left[g_{11}, \ldots, g_{nm}\right]\right] \qquad O = \mathtt{Zip(}\mathtt{f}_1, \ldots, \mathtt{f}_n\mathtt{)} \end{array}}{\mathtt{Let(}\left[\mathtt{f}_1 = o_1, \ldots, \mathtt{f}_n = o_n\right], \ \mathtt{in} \ O\mathtt{)} \ : \ T} \ \textsc{LetZip}$$

Figure 3.15: Projection Typing Rules

$$\frac{o_1 \ : \ T_1\left[\mathsf{V}_1\right] \qquad o_2 \ : \ T_2\left[\mathsf{V}\left[\mathsf{Index}\right]\right]}{\mathtt{Histogram(}o_1, \ o_2\mathtt{)} \ : \ T_1/\mathsf{A}^D\left[\mathsf{H}_{\mathrm{built}}\right]} \ \textsc{Hist}$$

Figure 3.16: Histogram & Partitioning Typing Rules

3.3 `MetaOps`: HiRes "Macros"

The primitive operators of HiRes are sufficiently low-level that most common relational operators a hypothetical query planner might wish to instantiate will decompose to a composition of several of them. For this reason it is convenient to have pre-made assemblages of them corresponding to tasks such as `Filter`, `HashPartition`, and `EquiJoin`, and a special layer on top of HiRes provides *meta-operators* for precisely those. They can be thought of as "macros" in the sense that they de-sugar to different HiRes-ASTs (depending on supplied tuning parameters), but are not type-checked until such expansion has taken place. Unlike, e.g., C macros or C++ templates, they are themselves designed to be pattern-matched, and re-written like terms of a relational algebra in their own right.

Since some of the transformations in Chapter 2 Section 2.3 involve multiple HiRes primitives, they can be made easier to implement by grouping those operators together before the transformation is applied. For example, the `Filter` meta-operator takes a list of filter predicates and can generate all the cracked and non-cracked variants of their conjunction, as well as the different possible collection and compaction schemes such a filter might enable. A partition meta-operator, in addition to bundling the separate histogram-building and reduction operations into a single unit, can generate different loop fusion choices across the boundary introduced by the blocking `Histogram()` and `Offsets()` operators. As a further example, the `Rename` meta-operator emits a HiRes `Project()` (or `Zip()`) primitive to extract only those record fields used by a later consumer, which can substantially simplify the expression of attribute packing.

The short table in Figure 3.17 summarizes the handful of existing meta-operators, which are used in all subsequent examples for the sake of concision. Compositions of operators can be built using the so-called "cake pattern" in Scala, in which each operator instance supplies methods to generate another operator of any type using the original as an input, resulting in multi-line chains such as that shown in the example of Listing 3.9. They also provide methods beginning `.withX(...)` to adjust tuning parameters `X`, and these also participate in cake pattern chains.

Meta-Operator	**Description**
`Concrete(`*Op*`)`	Contains an already-expanded HiRes operator
`Connector(`*MetaOp*, *Op=>Op*`)`	Insert arbitrary HiRes operator into expansion
`HashPartition(`*MetaOp, Id*`)`	Partitions with `Id` as key
`EquiJoin(`*MetaOp, MetaOp, Id, Id*`)`	Joins meta-ops with given keys
`Filter(`*MetaOp, Expr*`*)`	Filters input with list of predicates
`Aggregate(...)`	Either `Reduce()` or `HashTable()` aggregation
`Rename(`*MetaOp, (Id,Expr)*`*)`	Computes expressions with given field names

Figure 3.17: HiRes "Meta-Operator" Macros

```
 1 val q19meta: MetaOp = {
 2   import TpchSchema.{DELIVER_IN_PERSON, AIR, AIR_REG, [...]}
 3   val litem = Concrete(’lineitem_,  [...])      // Input both relations as ‘MetaOp‘s
 4   val part = Concrete(’part_, [...])
 5   var table: MetaOp = part                      // Build a hash table from this
 6   /* table = [...] */                           // Partition or parallelize table
 7   table = table.rename()                        // Prune unused fields
 8   var join: MetaOp = litem                      // Build a join operator ‘MetaOp‘
 9   /* join = [...] */                            // Parallelize join
10   join = join                                   // Pre-join filter
11     .filter(
12       (’l_shipinstruct === DELIVER_IN_PERSON),
13       (’l_shipmode === AIR
14         ’l_shipmode === AIR_REG))
15   /* join = [...] */                            // Partition probe side
16   join = join
17     .equiJoin(table, ’l_partkey, ’p_partkey)
18   /* join = [...] */                            // Customize join operator
19   val postCond: Expr = [...]                    // all post-join predicates
20   join = join                                   // Post-join filter & aggregation
21     .filter(postCond)
22     .rename(’price ->                           // Convert to double for stability
23       Cast(’l_extendedprice, lo.LoDouble()) *
24       (DoubleConst(1.0) - ’l_discount))
25     .aggregate((’price, PlusOp))
26     .nestedSumDouble(’price)
27   /* join = [...] */                            // Reduce parallel results
28   join.cast(UField(0), lo.LoFloat())            // Final result to return
29 }
```

Listing 3.9: A Plan Skeleton for TPC-H Q19

3.4 HiRes Case Studies

To demonstrate the utility of the HiRes language in expressing machine-level plans, this chapter concludes with case studies encoding TPC-H Q19 and other examples.

TPC-H Query 19

Listing 3.9 is a kind of "baseline" machine-level plan for TPC-H Q19 expressed as a HiRes meta-operator. It implements a serial variant of the plan depicted in Figure 2.11b, which fully packs both side of the join[11]. To generate the other plan variants, additional meta-operator code can be inserted at the positions marked by comments.

For example, the probe phase may be parallelized by inserting the following at Line 9:

[11] Although actually in this case the packing of `lineitem` occurs during the join, rather than before it

```
join = join.splitPar(threads)
```

And then adding the following reduction at Line 27, to complete the merge portion of the split-merge parallelism induced by `.splitPar()`:

```
join = join.nestedSumDouble(UField(0))
```

To partition the probe (`lineitem`) side, the following can be added at Line 15:

```
.partition('l_partkey, renamed = Some(_.rename()))
  .withHash(partHash)
  .withParallel(threads > 1 && !threadLocal)
  .withBlock(blockProbe)
  .withGather(!partAll)
```

This listing reveals other parameters that a plan *generator* might vary. First, `withParallel` determines whether the output of partitioning merges input threads' results, or whether each thread performs an independent partitioning. Second, `withHash` chooses the hash function to use for the partitioning and sets the number of radix bits. The `withBlock` modifier generates a `Cat()`/`Uncat()` pair to block the value (as opposed to key) input from being fused with the initial histogram build, in the case where this would result in sub-optimal materialization. Finally, `withGather` chooses whether the partitioning is packed or not: if not, then a subsequent `Gather()` will be inserted into consuming meta-ops to access fields other than the key. Joins take similar modifiers.

Using these tuning knobs, a *plan generator* can make the above insertions conditional up on variables set by an auto-tuner, in order to explore the performance characteristics of the broader plan space, as presented in Chapter 6. Generators must take care to insert artificial nesting (*shells*) on the build side prior to hash table construction when the probe side becomes deeply nested, as otherwise the buckets of the built table might get aligned with some level of nesting on the probe side, preventing each probe from selecting a bucket, as the loop induction variable for that nesting level would choose which bucket gets probed, rather than the key or hash function. This is fine in the case where both sides are partitioned at that depth based on the join key, but not for other kinds of nesting.

TPC-H Q06

The relative simplicity of TPC-H Q06–and its associated plan space–makes it easier to write a full plan generator as a Scala function returning different possible meta-operator configurations. Listing 3.10 shows the code for this in full, as it spans only a few lines! Its only machine-level planning parameters are (1.) split-merge parallelization, (2.) whether to crack the initial predicates, and (3.) whether to interleave collection of masked values.

```
def generateQ06Meta(
    threads: Int=1,
    crack: Boolean=false,
    collect: Boolean=false): MetaOp = {
  val meta: MetaOp = Concrete(’lineitem_,  [...])
  meta
    .splitPar(Const(threads))
    .filter(
      ’l_shipdate > minDate && ’l_shipdate < maxDate,
      ’l_discount > minDiscount && ’l_discount < maxDiscount,
      ’l_quantity < maxQuantity)
        .withCracked(crack).withCollect(collect)
    .rename(’revenue ->
      Cast(’l_extendedprice, lo.LoDouble()) *
      Cast(’l_discount, lo.LoDouble()))
    .aggregate((’revenue, PlusOp))
    .nestedSumDouble(’revenue, mute = threads < 2)
    .cast(UField(0), lo.LoFloat())
}
```

Listing 3.10: A Plan Generator for TPC-H Q06

Chapter 4

Operator Fusion: Constraints & Algorithms

4.1 Introduction

A primary benefit of query compilation is its ability to *fuse* multiple relational operators together, permitting the re-use of data while in registers or cache, and lessening the likelihood of communication between units of parallel execution.

Fusion is both a challenge, because the choice of how to carve up an operator DAG into fusible components is under-constrained by program semantics without being fully free (See Figure 2.8 in Chapter 2), and an opportunity, because different fusion options require varying amounts of data movement and present novel tradeoffs between computation and communication that may affect code generation today as well as database hardware design tomorrow. It is an aspect of machine-level query planing worthy of significant investigation, and this chapter describes its expression and manipulation in the HiRes compiler stack.

These concerns are not, strictly speaking, novel. Loop fusion is a well-known compiler optimization technique [15, 40] that, in the general case, identifies loop nests with conformable index variables that can be combined without violating data dependencies, and which can *contract* array values to scalars, reducing their storage and communication requirements. This study considers loop fusion in a slightly more specific context and with somewhat different constraints.

In order to expose fusion choices as a dimension of machine-level query planing, HiRes needs to supply a *consistent mapping between HiRes code and its resulting sequence of loop nests*, effectively making loop fusion decisions part of the language's semantics. Since another goal of its design was to facilitate the investigation of query optimization using, at least initially, handwritten plans, HiRes had to ensure that the final loop nest choice is *intuitive* from the programmer's perspective, and that it follows naturally from the structure of the code, even if other, less straightforward possibilities might be more optimal. The following sections thus do not seek to prescribe optimal decision strategies for fusion-conscious query planing, as this analysis is deferred to the communication-aware cost model discussion of Chapter 7.

The desire for an intuitive fusion algorithm also influenced the decision to perform fusion analysis at an early stage in the compiler pipeline, prior to the point at which any concrete loop nests have been issued or induction variables assigned. This, too, departs from prior work, in which the input has generally been assumed to be a C-like program (or its control flow graph) containing for-loops that the compiler is entrusted to recombine as part of a back-end optimization process of which the programmer may remain largely unaware. The approach of HiRes, by contrast, entails the inference of compatible loop nests based on the structure of the input operator DAG, and does not attempt to reconstruct this relationship from lower-level (LoRes) code that has already been generated.

The foregoing criteria made it difficult to reuse off-the-shelf algorithms, and motivated the formulation of new fusion procedures. The rest of this chapter outlines principles of "intuitive" fusion choices, and formalizes these requirements into a set of algorithms, concluding with a brief discussion of issues arising particular operators' semantics.

4.2 The Shared Clique Formation Algorithm

In HiRes, two separate kind of fusion take place, corresponding to the array and vector levels of relation structure, respectively. Each is subject to different conditions governing which nodes may start and end a *clique* to be fused. Array fusion–by which operations that can share the same "thread ID" induction variable are merged–is performed independently for each nesting depth in increasing order, and is interrupted by nodes that change depth (or perform nested "array reductions"). Vector fusion–by which intermediate results can be *immaterialized* to register-allocated scalars–is blocked by operations that must fully process their input vectors before returning an output (such as hash table builds and aggregations), or when an operation requires that one of its inputs be fully materialized prior to use (such as a gather, or hash table probe). What both share is a common algorithm for dividing a DAG into fusible cliques once such determinations have been made.

An Intuitive Overview

The *clique formation algorithm* (CFA) assumes as its input a DAG in which all nodes have been marked according to whether they start or end a clique (or neither or both), and returns a set of valid cliques that do not violate the starting and termination conditions and which are acyclic. Unlike prior work, which proceeds by labeling DAG edges according to whether they support or inhibit fusion, and then feeding the result into a black-box constraint solver, the CFA attempts to capture the natural intuitions a human programmer might have upon inspecting the operator DAG. In particular, it seeks to embody the following principles:

1. **Start from the inputs**: It is easier to understand fusion choices when they are made (roughly) in program order. Also, good query plans tend to filter records out early, so doing more before the first materialization will likely be better.

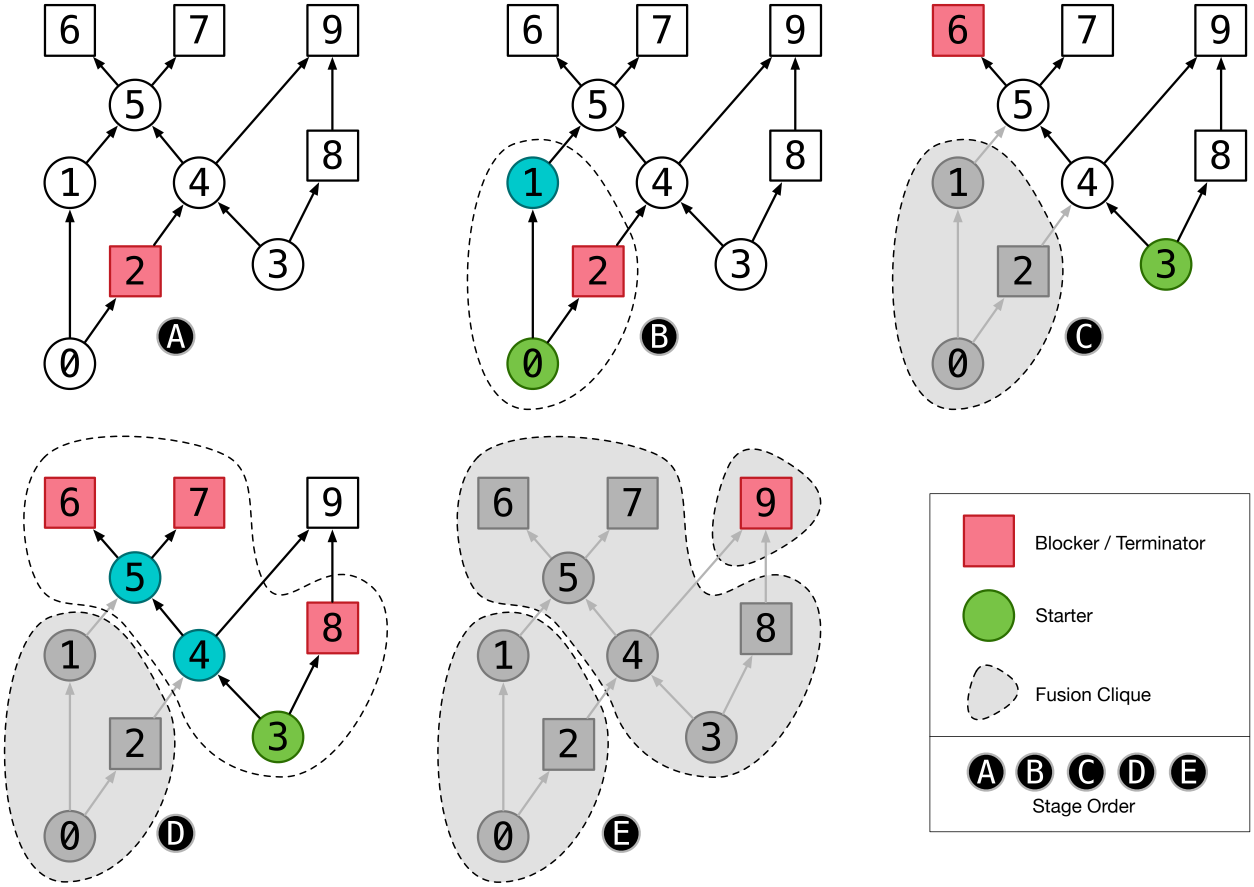


Figure 4.1: Execution of the Fusion Algorithm

2. **Pack cliques greedily**: Forming cliques one-at-a-time, and filling each maximally as it is considered, makes reasoning about the algorithm much easier.

3. **Process "leftmost" nodes first**: In multi-input operators, it is usually the syntactically first input whose structure determines the operator's loop induction variables (e.g. in a join, the left-hand side "probe" input causes probes to the hash table), so cliques should be built from left to right.

4. **Avoid forming cycles**: All correct fusion algorithms must not form any cyclic dependencies between fused cliques, but if cliques are formed one-at-a-time, then cycles can either be *split* when formed accidentally, or *avoided* because the algorithm will never result in them. Our solution is the latter, as Section 4.2 explains.

5. **Manual edits yield all variations**: Manual insertion of `Cat()`, and `Block()` nodes should be able to generate the other possible clique sets not chosen by the greedy algorithm.

Figure 4.1 shows how such an algorithm might process an abstract DAG once starter and terminator nodes have been marked (with circles and boxes, respectively). This could represent either inner or outer loop fusion, but here the choice is irrelevant. What matters is the sequence of steps from stage (A) to stage (E) in which three cliques are formed greedily. Nodes 0 and 3 (numbered in depth-first, left-to-right, post-order traversal from the output) are inputs to the HiRes program, and node 0 is the "leftmost" (first in program order), so it is a natural starting place. Intuitively, step (A) reveals that fusion is blocked by a nearby terminator (node 2), so in step (B), a clique is formed from the starter, the terminator, and any nodes dominated by the starter but not the terminator (node 1). Note that a node n_1 is *dominated* by another node n_2 ($n_1 \sqsubset n_2$) if n_2 is an input of n_1, or is an input of any other node that dominates n_1.

Searching from successive starters to their closest terminators would seem the most natural procedure. However, as step (C) shows, the HiRes CFA makes a slight tweak: it is actually the closest *undominated terminator* to the (leftmost) input that is considered to form the next clique. Here, *undominated* means that no other terminator not yet added to a clique is a dominator of the node in question. Thus, in step (C), the next undominated terminator is node 6, and the only starter that dominates it is node 3. In step (D), node 3's additional undominated terminators (7 and 8) are discovered iteratively, until a maximal clique is formed. Finally, in step (E), only node 9 is left, and forms its own clique. In this example, the greediness of clique packing results in node 1's inclusion in the first clique, though it would certainly be just as valid to include it in the second, larger one. To achieve that result, the programmer would insert an explicit blocking node between 0 and 1.

The ever-so-slightly counterintuitive modification of choosing cliques by their terminators still fulfills the *start from the inputs* principle, as undominated terminators are inherently "closer to the input" than others, but it also helps enforce the *avoid forming cycles* condition as well. To see why, consider the abstract DAG in Figure 4.2: there, a clique formed first from node 2 would find node 0 as a closest terminator, and in the next iteration node 3 would be selected as a dominator of node 0, at which point the clique would be deemed full, as node 4 cannot be added due to its domination of the previously-included node 0. However, such a clique would inherently cause a cycle, since it both depends on and is dependent on node 4. Choosing cliques instead by terminator resolves any need to detect and split up such a candidate with an irresolvable cycle, because another node or clique that might possibly form a cycle with a newly-formed one would have a terminator that dominates it, and so would have already been removed from consideration. Section 4.2 formalizes this explanation.

Formalizing the Algorithm

Algorithm 2 crystallizes the foregoing considerations into a concrete procedure. Specifically, Cliques(S, T, X, DAG) accepts a $DAG = (E, V)$, a set of starter nodes S terminator nodes T, and *excluded* nodes X which must never be part of a clique, and returns a set C of sets of all nodes inside each generated clique.

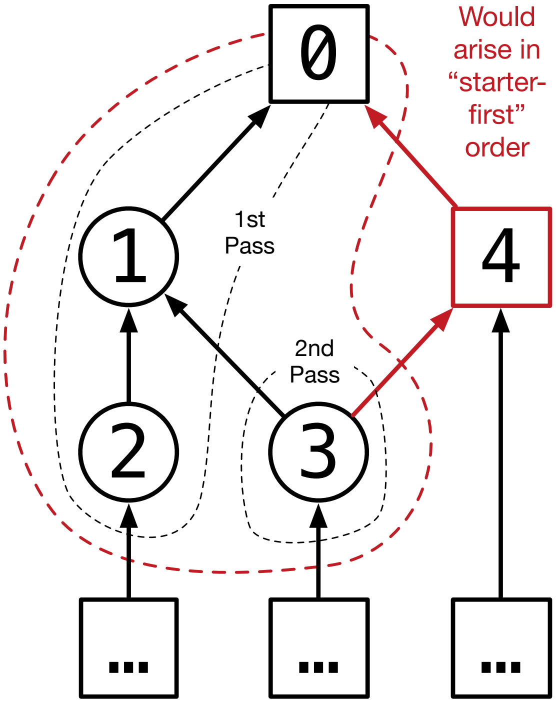


Figure 4.2: Cycles from "Starter-First" Greedy Clique Formation

Preconditions

The first precondition of Algorithm 2 is that all excluded nodes must be terminators: $X \subseteq T$. However, not all terminators must necessarily be excluded: aggregations, for example, may be contained within their inner loop fusion clique, but no node that depends on them may be. It it is necessary for all excluded nodes to be terminators, however, as they would terminate any clique to which they were adjacent.

Another precondition the caller of Cliques (an inner or outer loop fusion routine) must satisfy is that every node n which dominates a terminator $t_1 \in T$ must satisfy one of the following conditions:

1. It is a starter $n \in S$
2. Some starter $s_1 \in S$ dominates it $s_1 \sqsupset n$ and no terminator $t_2 \in T$ exists such that $s_1 \sqsupset t_2 \sqsupset t_1$
3. A terminator $t_2 \in T$ exists such that $t_2 \sqsupset t_1$ and $n \sqsupset t_2$

In other words, these conditions require that every terminator actually ends some potential clique, which is necessary to ensure forward progress, as explained below.

Otherwise, there are no restrictions on which nodes can appear in S, T, and X, aside those which follow from the semantics of the operators and are encoded by the caller; a node can be both a starter and a terminator, and these can be either included in or excluded from the cliques they form. Sections 4.3 and 4.3 explain how array- and vector-level fusion satisfy this requirement, and preserve language semantics in their formation of starter, terminator, and exclusion sets.

Postconditions

In return, Cliques(S, T, X, DAG) makes the following guarantees, which define correctness:

1. **Termination**: the algorithm will terminate, with runtime proportional to $\Omega(|T|)$.
2. **Acyclicity**: The resulting cliques will not contain any cycles
3. **Disjointedness**: Each node will be included in at most one clique

How it works

The CFA is stated in terms of *dominator tables* $D[n]$ which list for each node n the set of all nodes that are dominated by it, and *inverse dominator tables* $\bar{D}[n]$ that list all nodes dominating n. These are computed by the procedure in Algorithm 1, run once prior to to the main body of the CFA (Lines 3-4 of Algorithm 2).

Algorithm 1 Dominator Table

```
procedure DomSet($DAG$)                          ▷ For each node, all the nodes it dominates
    $D[n] \leftarrow \emptyset$ for $n \in DAG$                      ▷ Nodes dominated by $n$
    $F \leftarrow \{n : \mathsf{out}(n) = \emptyset\}$                 ▷ Frontier, starting at inputs
    $M \leftarrow \emptyset$                                    ▷ nodes that have been marked
    while $F \neq \emptyset$ do
        $M \leftarrow M \cup F$                                         ▷ Mark the frontier
        $F_{\text{next}} \leftarrow \emptyset$
        for $n \in F$ do
            for $m \in \mathsf{in}(n)$ do
                $D[m] \leftarrow D[m] \cup D[n] \cup \{n\}$
                if $\mathsf{out}(m) \subseteq M$ then                ▷ If all $m$'s children are marked
                    $F_{\text{next}} \leftarrow F_{\text{next}} \cup \{m\}$
        $F \leftarrow F_{\text{next}}$
    return $D$
```

Algorithm 2 then iterates until it has considered and removed every terminator in the set T, starting with the leftmost remaining, undominated terminator in each iteration. Once the clique-defining terminator has been chosen, the largest possible clique is built around it in an inner loop which gradually assembles *clique starter and terminator sets*, S' and T' respectively. The inner loop adds any starters that dominate the current terminators in T'

and aren't dominated by them, but which may be dominated by or dominate other starters. In the opposite direction, S' is expanded with any starters that dominate T', so long as they are not themselves dominated by any it. This process is repeated until no new starters or terminators are added.

At that point, a new clique has been found. Line 18 computes its interior as the set of all nodes that are dominated by at least one starter in S', end *not dominated by any terminator in* T'. This formulation results in the maximal clique size; were the second condition instead reversed to select all nodes that dominate at least one terminator in T', this could result in, for example, node 1 of Figure 4.1 being excluded from the first clique and included in the second. The clique interior I is further winnowed by the exclusion of any *marked nodes* M, which have already been included in another clique, and so must not be used, and by the excluded nodes X on Line 19.

Nodes are marked whenever included in a clique, but starter and terminator nodes must eventually be removed from consideration even if they can never be included in any clique. Starters may begin multiple cliques, but cease to matter when all their outputs have been marked, unless they are also terminators themselves (recall that some nodes can be both), in which case a separate condition for terminator removal applies. Excluded terminators should be marked only when all their inputs are either marked, or are excluded themselves. The latter condition ensures proper handling of the degenerate case where starters are joined directly to terminators, which otherwise would prevent termination of the algorithm. It is important to note the asymmetry of starter- vs. terminator-removal conditions: starters are not marked when their remaining outputs are all terminators, as they might themselves be terminators, and could be needed to find additional cliques; terminators do not have this constraint, as they are the ultimate delineators of new cliques.

Correctness Proof

The three correctness conditions in Section 4.2 may be proven separately, but first Lemma 1 simplifies some of the arguments.

Lemma 1 (Terminator Dominator Inclusion)**.** *If a non-excluded ($n \notin X$) node $n \sqsupset t$ dominates a terminator t of a clique C, and does not dominate any other terminator $t' \sqsupset C$ that dominates (any node in) the clique, then n is included in clique C.*

Proof. Since $n \sqsupset t$, then by Line 13, its closest dominating starter s_n would eventually be added to the clique if it was not initially. Moreover, there must *be* a dominating starter s_n because of the preconditions in Section 4.2, which require nodes n dominating a terminator t with no intervening terminator $n \sqsupset t' \sqsupset t$ to have, or be, a starter that dominates t. There could not be any terminator that precludes s_n's addition, since this would dominate n and, transitively, t, and would therefore have been already eliminated in a previous iteration. Then, since Line 18 includes all nodes dominated by at least one of the clique's starters, n will be included. □

Algorithm 2 Fusion Clique Formation Algorithm

```
 1: procedure Cliques($S, T, X, DAG$)    ▷ Global set of starters, terminators, excluded
 2:     $C \leftarrow \emptyset$    ▷ Set of cliques built
 3:     $D \leftarrow$ DomSet($DAG$)
 4:     $\bar{D} \leftarrow$DomSet(inv($DAG$))    ▷ DomSet of the inverted DAG
 5:     $M \leftarrow \emptyset$    ▷ Marked nodes removed from consideration
 6:     while $T \neq \emptyset$ do
 7:         $T', S', \leftarrow \emptyset$    ▷ Terminators & starters for next clique
 8:         $t, s \leftarrow \emptyset$    ▷ Next terms. & starters to add to clique
 9:         $T_{undom} \leftarrow \{n \in T : \bar{D}[n] \cap T = \emptyset\}$    ▷ Set of terminators not dominated in $T$
10:         $T', t \leftarrow$ {leftmost element of $T_{undom}$}    ▷ Starting terminator
11:         while $t \neq s \neq \emptyset$ do    ▷ Iterate until no new starters/terminators
12:             $s \leftarrow \bigcup_{n\in T'} \left(\bar{D}[n] \cap S\right) \setminus M$    ▷ Consider new unmarked starters
13:             $s \leftarrow \{n \in s : \nexists_{m\in T'\cup S'\cup s} n \in D[m]\}$    ▷ Only take if undominated
14:             $S' \leftarrow S' \cup s$
15:             $t \leftarrow \bigcup_{n\in S'} (D[n] \cap T) \setminus M$    ▷ Unmarked terminators dominated by $S'$
16:             $t \leftarrow \{n \in t : \nexists_{m\in t\cup T'} n \in D[n]\}$    ▷ Only take if undominated
17:             $T' \leftarrow T' \cup t$    ▷ Add them to terminator set
18:         $I \leftarrow \left(\bigcup_{n\in S'} D[n]\right) \setminus \left(\bigcup_{n\in T'} D[n]\right)$    ▷ Compute interior
19:         $I \leftarrow (I \cup T') \setminus M \setminus X$    ▷ Add terminators, but remove excluded
20:         $M \leftarrow M \cup I$    ▷ Mark all interior nodes
21:         $C \leftarrow C \cup \{I\}$    ▷ Add new clique to output set
22:         $S_{\mathrm{mark}} \leftarrow \{n \in S : \mathsf{out}(n) \subseteq M\} \setminus X$
23:         $M \leftarrow M \cup S_{\mathrm{mark}}$    ▷ Mark starters w/ all-marked outputs
24:         $T_{\mathrm{mark}} \leftarrow \{n \in T : \mathsf{in}(n) \subseteq (M \cup X)\}$
25:         $M \leftarrow M \cup T_{\mathrm{mark}}$    ▷ Mark terms. w/ all-marked inputs
26:         $T \leftarrow T \setminus M$
27:         $S \leftarrow S \setminus M$
        return C
```

Essentially, Lemma 1 expands the scope of Line 18 to include in a clique not just nodes dominated by one of its starters (as explicitly stated), but also those that dominate one of of its terminators, which is implied by the inner loop above. This relies on the precondition that terminators must have starters, which is necessary for semantic correctness as well.

Acyclicity

Theorem 1. *Algorithm 2 will not produce any cliques that result in a cycle*

Proof. Figures 4.3 and 4.4 show the two types of cycles that could potentially arise in clique formation, and how a non-cycle-avoiding CFA would have to resolve them. In the first case, a clique is formed that has an irresolvable *self-dependency* (Cycle Type I) through an external

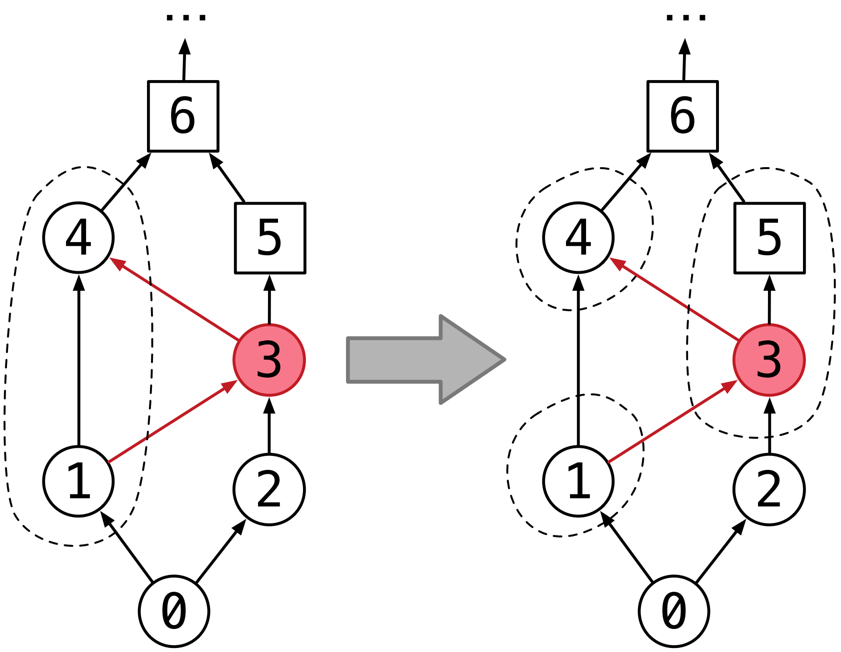


Figure 4.3: Cycle Type I: Unresolvable "Self-Dependency" Through Single Node

node (or nodes) that both depend on the clique and on which it is itself dependent. In the second case, a potential clique would form a cycle with a previously-formed clique (Cycle Type II), even if there is no reason why that candidate is inherently invalid due to a self-dependency–i.e., splitting up the previous clique would resolve the incompatibility. These two cases can be excluded separately, and for different reasons.

Lemma 2. *Algorithm 2 will not form any cliques with an irresolvable cycle through an exterior node or nodes (Cycle Type I).*

Proof. The Cycle Type I case of Figure 4.3 has two variants of its own, as Figure 4.5 depicts.

In Figure 4.5a, a clique candidate on the left would have a cyclic dependency on node 1 (without loss of generality to examples containing multiple nodes in node 1's place), but this would never result from the execution of Algorithm 2. The reason for this is that node 1, in forming a cycle with the clique, must necessarily dominate one of the clique's terminators, and since it was assumed not to be a terminator itself, by Lemma 1, node 1 would have been included in the clique, preventing a cycle. Thus, *forming the maximal greedy clique prevents this kind of cycle.*

The other Type-I case, shown in Figure 4.5b, is a cycle through a terminating node, which *cannot* be included in the candidate clique. This type of cycle could only arise if the proposed clique includes nodes dominated by the offending terminator in node 1, and therefore also has at least one terminator of its own that node 1 dominates. However, this cannot occur

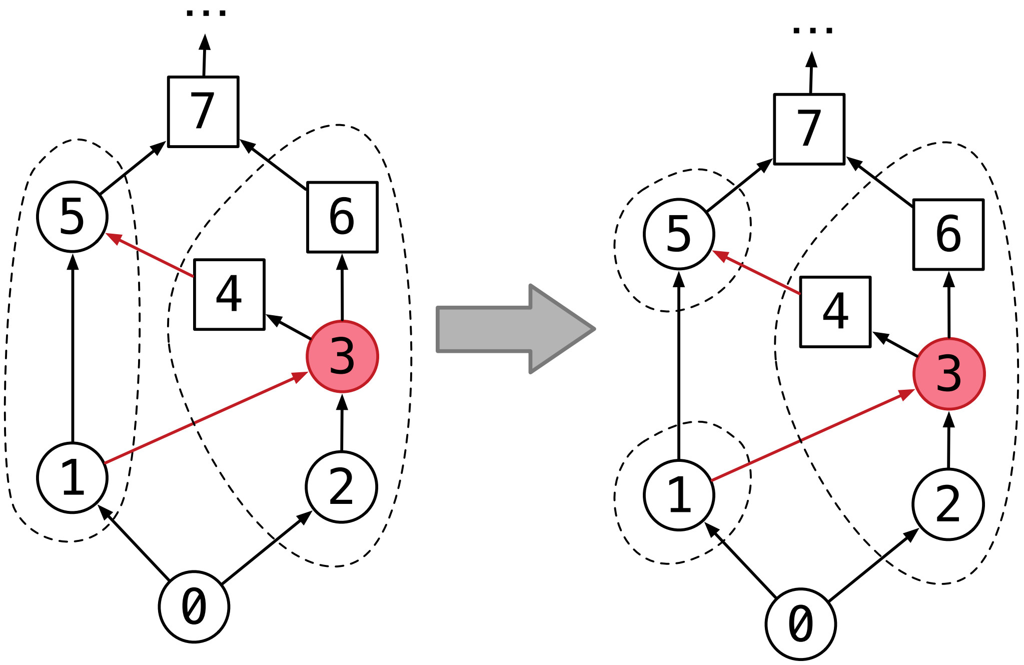


Figure 4.4: Cycle Type II: Mutually-Dependent Fused Cliques

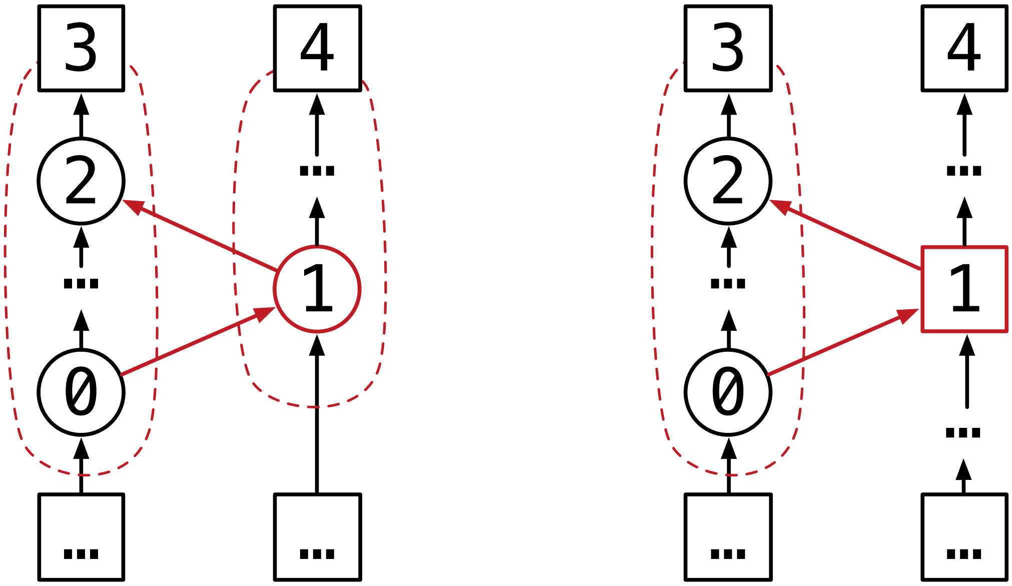
under the above algorithm, as Line 10 always chooses an *undominated terminator* to start the clique, and Line 16 only adds undominated additional terminators, so no clique inclusive of nodes dominated by node 1 would be considered before node 1 has been marked. If node 1 has not yet been marked, then, assuming without loss of generality that node 1 is the closest undominated terminator dominated by some nodes in the left-side candidate clique, those nodes would be included in a clique terminated by node 1, based on the same reasoning as applied in the case of a non-terminated Type-I cycle. □

Lemma 3. *Algorithm 2 will never generate a clique that forms a cycle with a previously-formed clique (Cycle Type II).*

Proof. Assume the opposite: there exists some clique C_1 that would conflict with candidate clique C_2. Then there must be some node $n_2 \in C_2$ which dominates at least one node $n_1 \in C_1$. However, this node n_2 would have already been included in C_1, because it dominates n_2 and therefore also some terminator t of C_2, and would have been included in C_2 according to Lemma 1. □

Thus, by Lemmas 2 and 3, the clique formation algorithm does not result in cycles.

QED



(a) Without Terminators

(b) Through a Terminator

Figure 4.5: Two Kinds of Type-I Cycles

Termination

Algorithm 2 can be shown to terminate after a number of iterations of the outermost while loop that is asymptotically proportional to the number of edges, but note that its actual runtime, as presently stated, will by quadratic in the number of nodes. Algorithm 4 remedies this inefficiency.

Lemma 4. *The leftmost undominated terminator t_L is always marked (removed) in each iteration of the clique formation algorithm.*

Proof. If t_L is not in the excluded set X, then it will be marked because it is included in the generated clique by Line 19. On the other hand, if t_L *is* excluded, it will also be marked, because all of its inputs will have been marked or are starters: if not starters, these inputs dominate t_L and would be included by Lemma 1. □

Theorem 2. *Algorithm 2 terminates in at most $|T|$ iterations.*

Proof. Since by Lemma 4 the leftmost undominated terminator will always be marked in each iteration of the CFA's outer loop, it will run for at most $|T|$ iterations. □

Finally, the number of iterations of the inner loop is at least loosely bounded by $|T|+|S|$, because each iteration either adds at least one new starter node or one new terminator, or exits, so the algorithm is certain to terminate. However, the preceding formulation in terms of dominator sets requires both their computation and use in the inner loop, which is of quadratic complexity in the number of nodes. This deficiency can be remedied by replacing the set-based operations of the inner loop with explicit breadth-first traversals, at which point the runtime of the inner loop is also more readily analyzable.

Disjoint Cliques

Theorem 3. *No node will ever be included in more than one clique.*

Proof. If a node $n \in X$ is marked as excluded, then it will not be included in any clique, because Line 19 explicitly removes it. If on the other hand $n \notin X$, then it is either included in no clique, or it is included in one, at which point it is marked. Line 19 prevents any marked nodes from being included in a future clique. □

An Improved CFA ($O(|E|)$ Runtime)

The formulation of Algorithm 2 in terms of dominator sets makes it easier to state, but unfortunately results in runtime that is quadratic in the number of DAG nodes. This can be reduced to asymptotically linear time in the number of DAG *edges* the replacing the inner loop with a sequence of breadth-first traversals that "discover" starters from terminators and vice versa.

Algorithm 3 generalizes the starter- and terminator-delimited BFS procedure needed by the revised CFA. It essentially "folds in" the BFS previously used to build the dominator table into the inner loop, ending it as appropriate when a terminator is encountered.

Algorithm 3 Terminator-Delimited BFS

```
procedure TermBFS($F, T, DAG, M, \textsc{inv}$)          ▷ All nodes between $F$ and $T$
    $F_{\text{next}} \leftarrow F$
    while $F_{\text{next}} \neq \emptyset$ do
        $M \leftarrow M \cup F$                              ▷ Mark the frontier
        for $n \in F$ do
            for $m \in \mathsf{out}(n)$ do
                if $\textsc{inv} \wedge m \notin M$ then
                    $F_{\text{next}} \leftarrow F_{\text{next}} \cup \{m\}$
                if $\neg\textsc{inv} \wedge \mathsf{in}(m) \subseteq (M \cup T) \wedge m \notin M$ then
                    $F_{\text{next}} \leftarrow F_{\text{next}} \cup \{m\}$
        $F \leftarrow F_{\text{next}}$
    return $M$
```

TermBFS(F, T, DAG) returns all the nodes *between* a frontier F and terminator set T–that is, all the nodes whose inputs are completely dominated by F and not dominated by T. This can be used to find new terminators from a set of starters, or, with the inverted DAG inv(DAG) and the "invert" flag inv set, to find new undominated starters from a frontier of terminators. In the latter case, however, the behavior is subtly different: the requirement that all of a node's inputs (in the reverse direction–i.e. outputs) be marked is dropped, because this allows new interior or starter nodes to be found that may be connected outside of the current terminator set.

However, correctness is still maintained, because no dominating terminator will be added by mistake, as any terminator that dominates nodes being considered for a current clique would have been marked already by a previous iteration of the outer CFA loop. In other words, the inv flag's special behavior reproduces the effect of Line 18 in the original CFA of Algorithm 2, which includes all nodes that are dominated by a starter and not by a terminator of the current clique (rather than just those that *also* dominate a terminator). The "forward" direction of traversal from S' to T still requires that all a node's inputs be marked before the node itself may be, as this prevents a node from being included in the clique until it is known for sure that it is not dominated by one of the clique's terminators.

In a revised clique formation shown in Algorithm 4, all lines involving dominator tables have been replaced by the two TermBFS calls in the inner loop that traverse the DAG in opposite directions.

Runtime of Algorithm 4

It simply remains to be shown that this revised algorithm has $O(|E|)$ runtime for a DAG with $|E|$ edges. Here, "visiting a node" means checking whether all of its incoming (or outgoing) edges are marked so that it may be added to the next frontier.

Lemma 5. *Algorithm 3 (*TermBFS*) visits each edge in the the unmarked DAG ($DAG \cap M$) exactly once.*

Proof. In the inverse direction, each node is added to the marked set soon as it is seen, so any edge that were visited twice would have to contain at least one node that was visited twice, which is not possible. In the forward direction, a node will be visited at most as many times as the number of edges flowing into it, or less if some of the edges come from unmarked terminators that dominate it. □

Theorem 4. *The runtime of Algorithm 4 is $O(|E|)$ for $DAG = (V, E)$.*

Proof. By Lemma 5, each call to TermBFS will visit each edge of the nodes in its result set M once per call. These returned nodes are immediately added to the global marked set, which will be passed to all subsequent calls, preventing them from being visited again. If in the worst case each node is visited by its degree, then the total number of visits is bounded by $O(|E|)$. □

Algorithm 4 Revised Clique Formation Algorithm

```
procedure CLIQUES2($S, T, X, DAG$)          ▷ Global set of starters, terminators, excluded
    $C \leftarrow \emptyset$                                    ▷ Set of cliques built
    $M \leftarrow \emptyset$                  ▷ Marked nodes removed from consideration
    while $T \neq \emptyset$ do
        $T', S', \leftarrow \emptyset$                  ▷ Terminators & starters for next clique
        $t, s \leftarrow \emptyset$                  ▷ Next terms. & starters to add to clique
        $T_{undom} \leftarrow \{n \in T : \bar{D}[n] \cap T = \emptyset\}$      ▷ Set of terminators not dominated in $T$
        $T', t \leftarrow$ {leftmost element of $T_{undom}$}               ▷ Starting terminator
        $I \leftarrow \emptyset$                              ▷ Start with an empty interior
        while $t \neq s \neq \emptyset$ do          ▷ Iterate until no new starters/terminators
            $i \leftarrow \textsc{TermBFS}(F = T', T = T, \textsc{inv}(DAG), M = I, \textsc{inv} = \mathsf{true})$
            $S' \leftarrow S' \cup i \cap S$
            $i \leftarrow \textsc{TermBFS}(F = S', T = \emptyset, DAG, M = I)$              ▷ $T$ is irrelevant
            $T' \leftarrow T' \cup i \cap T$
            $I \leftarrow I \cup i \cap X$          ▷ Expand interior with any new non-excluded nodes
        $M \leftarrow M \cup I$                                  ▷ Mark all interior nodes
        $C \leftarrow C \cup \{I\}$                              ▷ Add new clique to output set
        $S_{\text{mark}} \leftarrow \{n \in S : \mathsf{out}(n) \subseteq M\} \setminus X$
        $M \leftarrow M \cup S_{\text{mark}}$                  ▷ Mark starters w/ all-marked outputs
        $T_{\text{mark}} \leftarrow \{n \in T : \mathsf{in}(n) \subseteq (M \cup X)\}$
        $M \leftarrow M \cup T_{\text{mark}}$                   ▷ Mark terms. w/ all-marked inputs
        $T \leftarrow T \setminus M$
        $S \leftarrow S \setminus M$
    return C
```

4.3 Inner and Outer Loop Fusion Particulars

While both inner loop (i.e. vector) and outer loop (i.e. array) fusion share the CFA of Algorithm 4[1], they differ their selection of terminator and excluded nodes. This section describes how each of the two fusion types performs this selection while conforming to the CFA preconditions of and preserving *semantic correctness*, which means that fused nodes really should share a common loop induction variable (*cursor*) at the level of fusion, and that the terminator and excluded node choices obey the semantics of the particular operators they encompass. One commonality, however, escapes the previous section: it turns out, as shown below, that the very notion of starter nodes (S) is superfluous, and that, in fact, the preconditions of Section 4.2 *can only be satisfied if every node is a "starter"*. Thus, each fusion type must rely on the exclusion set X alone to achieve semantic correctness, and the clique formation algorithm can be simplified once more.

[1] The current compiler prototype implements the slower set-based algorithm

All Nodes Are Starters

The main reason for this is the requirement that every terminator must have a dominating starter–the second precondition in Section 4.2. This was necessary for forward progress, as its absence would void the result of Lemma 1, and prevent the incorporation of a maximum number of nodes into each candidate clique.

Theorem 5. *Every node $n \in V$ of a $DAG = (V, E)$ is a starter: $S = V$.*

Proof. If there were nodes $n \notin S$ in the DAG that are undominated by any $n \not\sqsubset s \in S$, then these could not dominate any terminator $n \sqsupset t \in T$, or else they would not be able to satisfy the above condition. This situation would easily arise when excluded nodes $x \in X$ are used by other excluded nodes $x' \in X$, where the excluded set X is defined strictly by semantic conditions such as materialization or reduction. Since in this case $x \sqsupset x'$, and since $X \subseteq T$, this arrangement violates the starter domination criterion, as x dominates a terminator. □

In addition to being necessary, the promotion of all nodes to starters actually simplifies both kinds of fusion, which now merely need to distinguish between nodes that terminate, but may be fused, those which must be excluded, and those with no constraints. It also simplifies the algorithm once more, this time by eliminating the separate tracking of starters outside of the inner loop; inside the inner loop, "starters" are simply those closest to the inputs of an a clique candidate being formed.

Outer Loop (Array) Fusion

Of the two fusion types, outer loop fusion is the more complex, as it attempts to unify nodes whose loop nests must have conformable induction variables at possibly multiple different levels. Two cursors conform when they have the same upper bounds, and this quality can be inferred implicitly from the structure of the DAG. In a one-dimensional DAG, where no node has more than one input, cursor conformation is simply a matter of "parsing" the nesters and flatteners that have been encountered in traversing the DAG from input to output: if N nesters have been seen at node n_1, and n_1 does not change nesting, then its cursors at array depths $1 \ldots N$ must have the same bounds as those of subsequent node n_2; if n_3 is a flattener, then n_4 cannot conform to n_1.

Multi-input nodes complicate this inference. When an operator accepts multiple nested inputs, it is generally implied that each dimension (depth) of each input contains the same number of elements as its corresponding depth in each other input (though this is not presently enforced by the type checker, as Chapter 3 explains), which can only be true if all inputs also have the same degree of nesting, although they may contain vector bases of differing lengths, or a mixture of scalar and vector bases. Nodes with multiple inputs would thus seem to require equal numbers of as-yet unflattened nesters along the path to each of them, making the "parsing" problem inherently two-dimensional. This raises, in turn, two questions: first, how can this 2-D analysis be performed, in particular when all nodes must be marked (implicitly) as starters, and second, must nodes with multiple inputs of differing depths be excluded from fusion altogether?

Iterating Over Depths

The answer to the first question is that outer loop fusion is performed *iteratively over each possible nesting depth*, starting from depth 1 onwards. At each depth, the DAG is first analyzed to mark all nodes according to whether they have on their input paths a sufficient degree of nesting to reach that depth, and all nodes that do not are marked in the exclusion set (and thus also as terminators). Once clique formation decisions have been made for depth N, the resulting cliques are then processed independently with a *recursive application of the outer loop fusion algorithm* with the target depth set to $N+1$. Performing outer loop fusion iteratively at increasing depths ensures that nodes will only fused at a given depth when their cursors at all lower depths have been shown to be conformable by virtue of having been fused. Furthermore, this makes outer loop fusion a prerequisite to inner loop fusion: if two nodes cannot be part of the same outer loop, they clearly cannot be merged into the same inner loop.

Depth Balancing

The answer to the second question is that nodes with inputs of initially differing depths *can* form a clique together, as this is made possible by the same mechanism needed to implement the *broadcast semantics* of Chapter 3, Section 3.2. When a node receives inputs of depth N and $N + 1$ simultaneously, a special *shell node* is inserted between it and the depth-N input. The shell node performs no operation but instead adds a degree of "fake" nesting that simply broadcasts (repeats) the depth-N array across all elements of depth $N + 1$. Afterwards, both inputs to the multi-input have the same depth, allowing it to fuse with the pre-existing depth-$(N + 1)$ input's clique. In the more general case, as many shell nodes as necessary are inserted to nest an node's unequal inputs up to the depth of the maximum, as indicated in Algorithm 5. Of course, there are exceptions for particular operators, whose semantics require or respond to nesting depth mismatches. These are explained below.

Preserving Operator Semantics Under Outer Loop Fusion

Array fusion excludes nodes that lack sufficient nesting, but must also exclude, or handle specially the following operator types:

- **`NestedReduce()`**: This operator terminates (but is not excluded from) array fusion *at its output depth* (but not below), as it reduces across all array elements
- **`LastArray()`**: While not strictly necessary, this flattener is treated as a nested reduction internally by the compiler, as one level of loop nesting must be skipped when its code is generated.

- **`Offsets()`**: Computing an offsets vector that merges N levels requires interrupting the N innermost array levels, as each array's offsets depend on all those that came before it.

- **`HashJoin()`**: In its simplest form, the hash-join operator expects a nested array as the hash table input, as the hash function selects the innermost array subscript as the bucket number (see Chapter 3, Section 3.1), and adds a layer of nesting to the output if the probe input is flat, making it a nester. However, if the probe input is already nested, it merely maintains the existing depth, and does not impact array fusion.

- **`Cat()` and `Uncat()`**: Concatenation is provided as a mechanism to force alternative clique formation decisions (see Chapter 3 Section 3.1), so it too blocks fusion at its output depth.

Algorithm 5 Shell Node Insertion

```
procedure InsertShells(DAG)
    $F \leftarrow \{n : \mathsf{in}(n) = \emptyset\}$                ▷ Frontier, starting at inputs
    $M \leftarrow \emptyset$                                          ▷ Marked nodes
    while $F \neq \emptyset$ do
        $F_{\text{next}} \leftarrow \emptyset$
        $M \leftarrow M \cup F$
        for $n \in F$ do
            $d \leftarrow \max\{\textsc{depth}(m) : m \in \mathsf{out}(n)\}$
            for $m \in \mathsf{out}(n)$ do
                $k \leftarrow m$
                for $i = \textsc{depth}(m) \ldots d$ do
                    Insert new shell $s_i$ between $n$ and $k$
                    $k \leftarrow s_i$
                if $\mathsf{in}(m) \subseteq M$ then
                    $F_{\text{next}} \leftarrow F_{\text{next}} \cup \{m\}$
        $F \leftarrow F_{\text{next}}$
```

Inner Loop (Vector) Fusion

Inner loop fusion must be performed last, after all outer loop fusion, and only those nodes which have successfully been fused at all outer nesting levels may be considered for inner loop fusion; otherwise, they would necessarily be in separate loop nests.

Some operators are *blocking* operators, as they must consume a whole vector of input elements before producing any output, and so interrupt inner loop fusion, while other operators must produce complete *outputs* before their dependent nodes may make use of them:

Algorithm 6 Array Fusion

```
procedure ArrayFuse(DAG, d)                       ▷ Array fusion for starting depth $d$ of a DAG
    $F \leftarrow \{n : \mathsf{in}(n) = \emptyset\}$                        ▷ Frontier, starting at inputs
    $M \leftarrow \emptyset$                                            ▷ Marked nodes
    $X \leftarrow$ {Nodes excluded by operator semantics}
    $T \leftarrow$ {Nodes that must terminate fusion at this depth
    while $F \neq \emptyset$ do                         ▷ Build starter and terminator sets
        $M \leftarrow M \cup F$                                   ▷ Mark the frontier
        $F_{\text{next}} \leftarrow \emptyset$
        for $n \in F$ do
            if depth$(n) < d$ then
                $X \leftarrow X \cup \{n\}$                ▷ Exclude nodes with insufficient nesting
            if $\mathsf{in}(n) \subseteq M$ then
                $F_{\text{next}} \leftarrow F_{\text{next}} \cup \{n\}$
        $F \leftarrow F_{\text{next}}$
    $T \leftarrow T \cup X$                              ▷ All excluded nodes must terminate
    $C \leftarrow$ Cliques2$(T = T, X = X, DAG)$
    for $c \in C$ do
        Actually do fusion for clique $c$
        ArrayFuse$(c, d + 1)$               ▷ Do fusion at next depth *only* for fused cliques
```

- **`Reduce()`**: The (scalar) result of a reduction is not valid until all its inputs have been processed

- **`HashTable()`**: Hash tables are essentially reductions, and must be fully-built before use

- **`Histogram()`**: Histogram-building is also a sum-based reduction

- **`HashJoin()`**: Joins do not require materialization, but the previous versions of the compiler could not incorporate their additional probe loop into a fused inner loop, as the probe loop needs to absorb all subsequent inner loops of nodes that depend on it into its body. This has recently been fixed, but complicates the implementation of fusion.

- **`Partition()`**: Partitioning is effectively a scatter operation, and so although each output element is technically valid for use as soon as it is "moved", it is simply impractical to make use of outputs by dependent operators in any loop nest presently generated by the compiler.

- **`Collect()` and `Compact()`**: Collection changes the output *cursor* (position in the output vector), re-aligning the index of output elements relative to their corresponding inputs. Similarly to `HashJoin()`, there is no inherent reason why this could not be accommodated eventually, but at present all nodes in an inner loop fusion clique are require to have a common inner loop cursor.

Finally, the `Gather()` operator has a special requirement that its second source–the buffer from which indices are gathered–be material, so that arbitrary indices may be read from it at random. This would not be possible if it existed only as a single element at a time, as it would if it were produced within the same inner loop as the vector of indices to be collected. However, this is generally unlikely to arise in practice, since the producer of a gathered vector is unlikely to have an input cursor that is conformable to that of the index array, which often will be of a different length or at least driven by a different scan.

Chapter 5

Ressort Compiler Architecture & Implementation

Ressort compiles HiRes into C++. This entails turning an implicitly typed, parallel data-flow graph of operators into an explicitly typed sequence of imperative loop nests that use only the set of basic operators defined by the C++ language. The compiler does so by progressively refining HiRes through a series of increasingly low-level intermediate representations, upon each of which different optimizations and transformations are performed.

A key choice underpinning the design of the Ressort compiler was to use C++ with parallel loop annotations as a compilation target. This wasn't strictly necessary, but neither was it unusual for a research query compiler, as several concurrent works have demonstrated [9, 60, 68]. At one extreme, Ressort could have targeted machine language directly, but this would have required an extraordinary amount of effort re-create the functionality of a standard compiler backend, which is both out of scope and also unnecessary, given the availability of existing, high-performance, open-source tool-chains. Many compilers for novel languages (as well as query compilers [25]) have been built on top of the LLVM framework [31], which supplies machinery for the lowest level of compilation tasks such as register allocation, code generation and scheduling, low-level optimization, and everything else needed to produce an executable.

This was also rejected, for two reasons. First, any translation from HiRes to LLVM-IR would necessarily transition through a loop-nest-like format, as the foregoing presentation of the language's semantics would imply, and the discussion that follows hopefully will elucidate. Lowering this to LLVM-IR, whose control flow graph (CFG) format comprises basic blocks of assembly-like instructions, would take unnecessary effort. That, after all, is the job of a C or C++ compiler's front end, meaning that it is significantly easier simply to leverage an existing tool for that purpose. Other query compilers have rejected this shortcut on the grounds that it increased compilation time too much [25], but latency was not the primary concern of this throughput-focused study.

Second, LLVM-IR lacks any explicit representation of fine-grained parallelism, meaning that Ressort would also need to either replicate the work of an OpenMP compiler, which translates parallel loops into functions to which work is dispatched, or supply a new alternative runtime library to implement task queues and other work-sharing constructs, whose utilization would still require lowering from loop-parallelism to some other form. It thus seemed more sensible to forego these efforts and once again leverage well-known tool-chains like GCC [14] or LLVM's own CLang.

However, outsourcing low-level compilation and optimization work to external packages is a double-edged sword. While reducing the compiler's development effort substantially, it also denies Ressort access to the results of common optimizations performed on IR codes prior to instruction selection and assembly generation. Some of these, such as constant propagation, common-subexpression elimination, and dead-code elimination, play a critical role in the refinement of compiled HiRes into a form suitable for some kinds of analysis, and debugging. Section 5.1 describes how the architecture of the HiRes compiler frontend results in redundant and difficult-to-read code fragments that also obscure the true relation between ostensibly different array subscripts and the real memory locations to which they refer. This untidiness also can also degrade performance, since GCC and LLVM backends generally cannot perform the relevant optimizations across parallel loop boundaries [58], once these have been lowered into independent procedures.

To enhance the utility of this output, then, some of the traditional compiler passes described above have been re-created within the Ressort framework. This is made possible by Ressort's own intermediate representation, called LoRes and described more fully in Section 5.2. LoRes is a fully typed intermediate language with a syntax closely resembling that of C, but which also preserves OpenMP-style parallel loop annotations and was designed to facilitate code generation from HiRes. In fact, some of the transformations applied to LoRes code are necessary parts of the HiRes compilation process, as they implement part of the language's semantics. Because these compiler passes now apply to typed, block-scoped ASTs, rather than to an LLVM-like CFG, the standard algorithms implementing them must be adjusted slightly, as this chapter explains.

The incorporation of LoRes as an IR makes Ressort a two-part compiler framework with a structure depicted in Figure 5.1. The first part transforms HiRes into LoRes while the second prepares LoRes for C++ generation, analysis, or user output. The remaining two sections of this chapter describe both compilation processes, respectively, and their interactions.

5.1 HiRes Compilation Pipeline

A key principle of the HiRes compiler is to separate operator-specific code generation from details of data layout and loop structure as much as possible. The implementation of `Eval()` should not depend on whether its input is a deeply nested array, whether the position of its next extent of records is computed by a simple offset or an indirect lookup, or whether those records are arranged in row-major, column-major, or some order in between. Of course, it is not possible to orthogonalize these concerns fully, as some operators like `HashJoin()` require one input to be nested, and must choose which vector-valued array element to process dy-

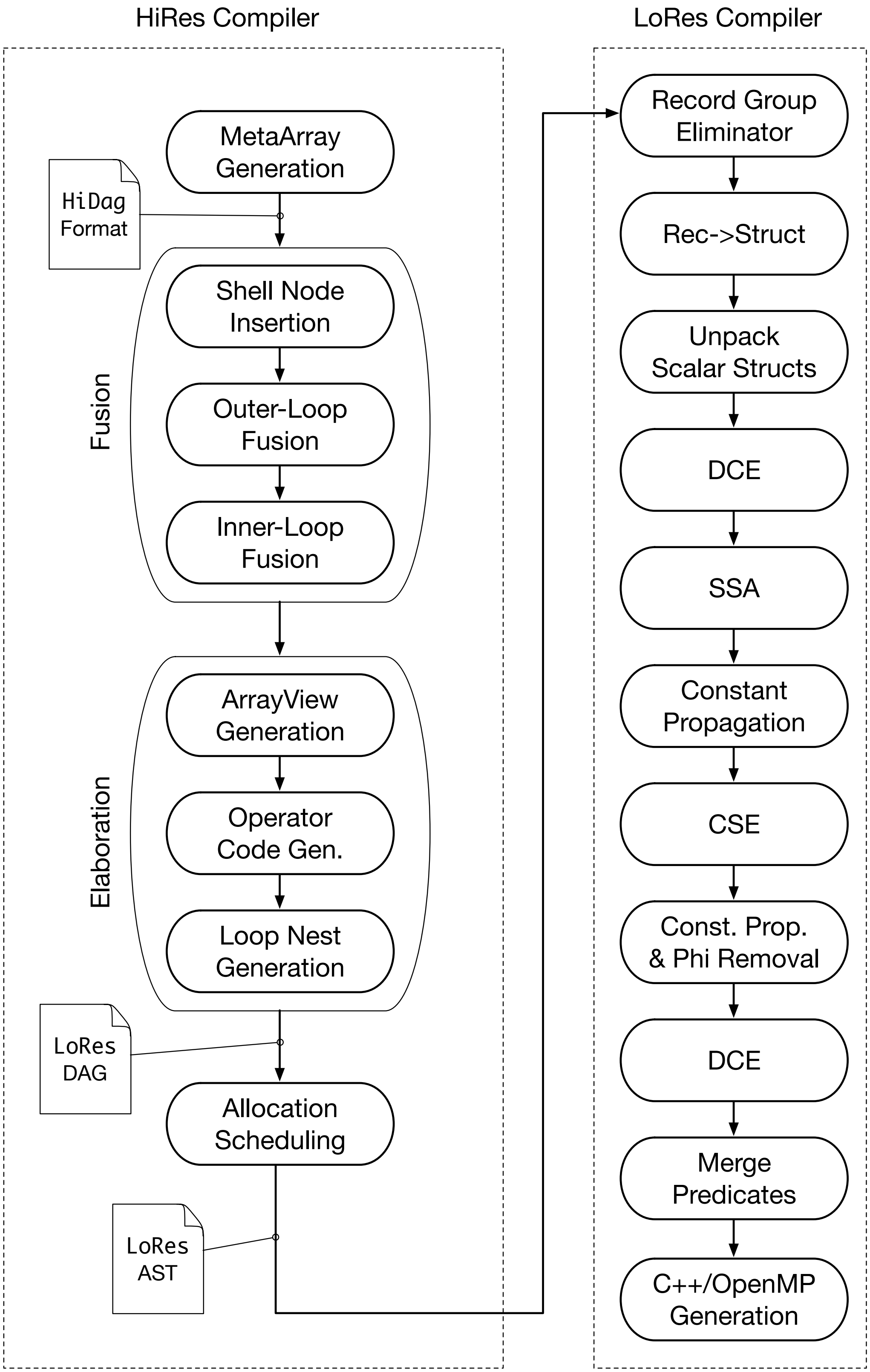


Figure 5.1: The Full Ressort Compiler Pipeline

namically, or may require, like `HashTable()`, access to type-specific operations ("append"), while others, like `SplitPar()` or `Zip()`, exist for the sole purpose of changing the shape of their inputs. At the very least, then, each operator's implementation should limit its concern to those aspects of structure that it requires or imposes. This is made possible by the sequence of intermediate abstractions through which HiRes is refined to LoRes along the left half of Figure 5.1.

Allocation: Meta-Arrays & HiDags (Abstraction #1)

After a HiRes AST is type-checked, it is split up into a DAG of operators, and each node's output value is represented as *meta-array* that augments its original HiRes datatype with other structural information such as the length of its underlying buffer(s), which it may or may not re-use from its inputs (pass-by-reference). Sizes, slice counts, and other such data may be supplied as constants, or may be known only dynamically, and so are encoded as LoRes expressions, as shown in Figure 5.2.

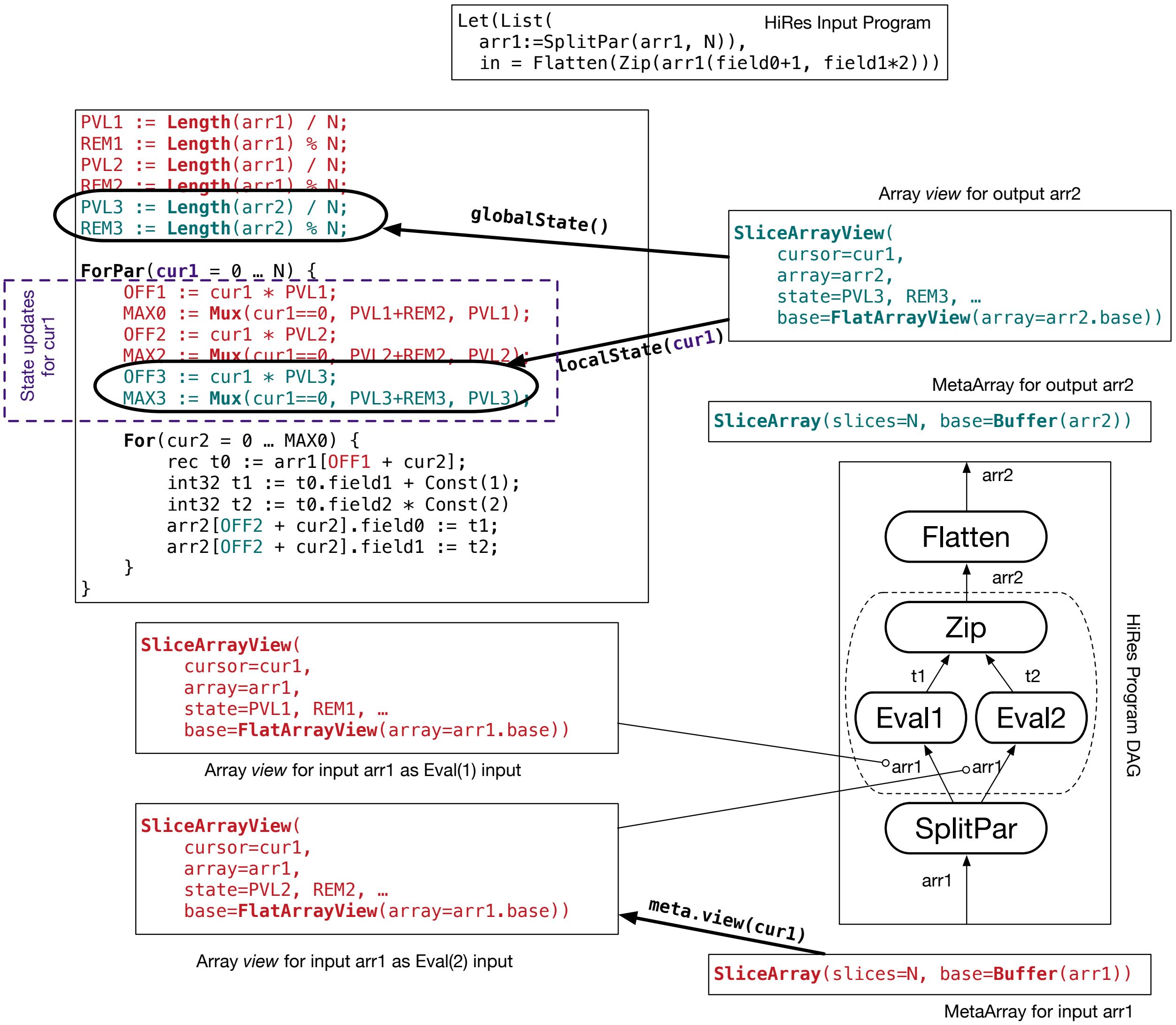
A meta-array encapsulates all the details needed to derive a new meta-array for a node's output based on its inputs, and exports a `clone()` method as part of its API that an operator-specific *allocator* can call to replicate its structure while adjusting only the attributes, such as length or record type, that it might change. Notably, when shape-changing operators such as `SplitPar(..., disjoint=true)` are applied, they do not always immediately result in the allocation of new output buffers with the specified disjoint structure (Figure 3.8); instead, the output meta-array is marked so that its `clone()` will produce a disjointed array only when a subsequent operator eventually requests a new buffer allocation. For example, in Figure 5.2, both the initial input, and the output of `SplitPar()` share the physical buffer `arr1`, even though they have different meta-array structures on top of it.

As a reflection of this intent to expose only the relevant layer of array structure to the allocator, meta-arrays are arranged in inverse order of their corresponding HiRes types: if an operator's output has type $\mathsf{A}^{C}\left[\bar{\mathsf{A}}\left[\mathsf{V}_1\right]\right]$, then its output meta-array structure would be:

```
SlicedArray(
  base =
    ChunkArray(
      base = Buffer(...),
      pointers = ..., ...),
  slices = numSlices,
  numValid = Some(...))
```

Only the outermost meta-array level, which divides each chunk into `numSlices` parts, is of interest to subsequent operations, which may remove it, in the case of a flattening, or add another layer, in the case of a nesting operator, without having to examine any interior layers.

The *HiDag* intermediate format is a graph whose nodes consist of HiRes operators, their allocated meta-arrays, and a set of *cursors* for each depth of array nesting present in an



```
Let(List(
  arr1:=SplitPar(arr1, N)),
  in = Flatten(Zip(arr1(field0+1, field1*2)))
```

```
PVL1 := Length(arr1) / N;
REM1 := Length(arr1) % N;
PVL2 := Length(arr1) / N;
REM2 := Length(arr1) % N;
PVL3 := Length(arr2) / N;
REM3 := Length(arr2) % N;

ForPar(cur1 = 0 … N) {
    OFF1 := cur1 ∗ PVL1;
    MAX0 := Mux(cur1==0, PVL1+REM2, PVL1);
    OFF2 := cur1 ∗ PVL2;
    MAX2 := Mux(cur1==0, PVL2+REM2, PVL2);
    OFF3 := cur1 ∗ PVL3;
    MAX3 := Mux(cur1==0, PVL3+REM3, PVL3);

    For(cur2 = 0 … MAX0) {
        rec t0 := arr1[OFF1 + cur2];
        int32 t1 := t0.field1 + Const(1);
        int32 t2 := t0.field2 ∗ Const(2)
        arr2[OFF2 + cur2].field0 := t1;
        arr2[OFF2 + cur2].field1 := t2;
    }
}
```

```
SliceArrayView(
    cursor=cur1,
    array=arr2,
    state=PVL3, REM3, …
    base=FlatArrayView(array=arr2.base))
```

```
SliceArray(slices=N, base=Buffer(arr2))
```

```
SliceArrayView(
    cursor=cur1,
    array=arr1,
    state=PVL1, REM1, …
    base=FlatArrayView(array=arr1.base))
```

```
SliceArrayView(
    cursor=cur1,
    array=arr1,
    state=PVL2, REM2, …
    base=FlatArrayView(array=arr1.base))
```

```
SliceArray(slices=N, base=Buffer(arr1))
```

Figure 5.2: LoRes Loop Generation & Intermediate Representation Levels

input or output. Each cursor is a loop induction variable. When operators are fused at a particular nesting depth, their HiDag nodes are combined into a single container node, and the cursors for that depth are all set to the same value in each fused node, ensuring that eventual traversals of their meta-arrays will be synchronized at that depth. Additionally, each HiDag node may *own* one or more cursors, meaning that it will generate the loop for that depth; in a fused node, only the outer container node owns the cursor for the fused depth.

Codegen: Array-Views & LoDags (Abstraction #2)

After shell node insertion (Section 4.3) and both types of fusion have been performed at the HiDag level, the DAG is *elaborated*, first by operator-specific code generation and then by the formation of loop nests around the result. Code generation begins with another lowering step in which each node is assigned a set of *views* of the meta-arrays used as its inputs or output. A view is a particular traversal of a data structure that might appear at multiple places throughout the DAG, much like an iterator in other languages. For example, in Figure 5.2, both `Eval` nodes have different views for the same meta-array produced by the split.

Each contains all the state needed to compute where the next extent of records from its meta-array should be accessed in the innermost loop, and supplies a `localState(cur)` method to generate LoRes code that updates that state when its level's cursor is modified. The dotted purple box in Figure 5.2 encloses state updates for the depth-1 level array views of all three nodes inside the fused sub-DAG, for instance. They correspond to the simplest nested view, the `SlicedArrayView`, which, like its originating `SplitPar()` operator, divides its base array into `N` equal-sized vectors, with any elements in the remainder consigned to the first iteration of its controlling loop. Like its homologues for more complex structures, its state ensures that the next loop level can access the correct buffer region with merely an array pointer, offset, and index. Since the remainder and slice lengths are constant across all loop iterations, they are computed as `globalState()`, to be declared just prior to the cursor-controlling loop.

More generally, each meta-array's view type generates state update code that ensures no calculations are needed inside the innermost (vector) loop other than the mere use of base and bounds values, and these assignment statements are inserted into the LoRes output during loop nest generation. Array views usually correspond directly to a particular layer of loop nesting. When an array view is created for a HiDag node's input or output, it is therefore normally tied to the cursor assigned to that level's depth in the node's cursor set, and its state updates are generated with respect to that cursor, but sometimes another input may be used, such as when `HashTable` selects which bucket to probe based on a data-dependent value.

Each view is granted a set of fresh LoRes variables to which it assigns state computed independently of any other views that may be performing the same traversal, or computing the same fractions and remainders of the same array lengths. This results in potentially a great deal of redundancy, and significant bloat in the size of the resulting LoRes program:

if a HiRes DAG has n nodes with average degree e and average nesting depth d, then the number of array-view state declarations will be proportional to $O(ned)$; when fusion is applied, each fusion boundary introduces E new "virtual input" nodes inside the fused DAG, where E is the number of edges crossing the input side of the fused node. *This explosion of state declarations is the primary motivation behind the LoRes-level common-subexpression elimination (CSE) optimization of Section 5.2.*

Dead code elimination also proves highly effective in the wake of array view-based code generation, as views always declare the maximal amount of state that they could possibly use, even if in a particular instance much is not needed (e.g. even non-padded views always declare padding-related offset calculations). The following, more detailed descriptions of array view mechanics provide more insight into how such cruft accumulates. While it would be tedious to present in detail the operation of each type of view, two important and illustrative ones are described below: sliced views, because they are the simplest, and chunk array views, because they are the most complicated.

Sliced Array Views

Figure 5.2 sketches a sliced array's state in its most minimal form: a *physical vector length*, computed by dividing the *logical vector length* of its *base array view* by the total number of slices. Every nested array view type exports (LoRes expression-valued) physical and logical vector lengths–so called for the views they afford of an imaginary underlying HiRes vector–for such use by any views that divide it further. The sliced array's logical length then consists of the physical length minus any padding that it imposes. Even though the sliced view assigns equal numbers of elements to each slice (remainder notwithstanding), that value may vary globally across the whole of a larger structure if, for instance, the view's base exports different logical vector lengths across the iterations of its own loop.

These variables are actually *local* state, and are computed inside each iteration of the sliced view's loop. Its *global* state, computed just above, includes the *average physical length* of each slice, and the *remainder* of elements if the slice count does not evenly divide the base's logical length. Inside the loop, division and modulus are then unnecessary, as these have been precomputed; the first loop iteration simply adds the remainder to the average.

Chunk Array Views

Figure 3.7 shows the structure of Ressort's compact hash-table format, designed to minimize the number of cache lines which must be accessed, in the average case, to probe a bucket. This hash-table is realized concretely as a *chunked meta-array* type, which actually has two view types: one whose elements are the hash-table's buckets, and one whose elements are the physically discontiguous "chunks", arranged in linked lists, that form expandable storage of each bucket. Traversing a chunked meta-array yields a double loop nest, of which the base chunked view corresponds to the outer loop over buckets, and the view nested on top of it is aligned with the inner loop over each chunk. Because the base view essentially equates to a sliced view of the pre-allocated, contiguous array of fixed-length buckets, it is fairly simple in comparison to the outer view, to which most of the complexity of managing chunks is delegated.

The outer view's state makes use, potentially, of several *ancillary* structures, each of which has a meta-array and views of its own: the *pointers* array holds pointers for each bucket's linked list of supplementary chunks, allocated on-demand after the bucket's initial reservation in the base array is exhausted, and the *counters* array, in which the current number of elements in each bucket is stored. When a bucket is selected by the base view (in the outer loop), the outer chunk view's global state update examines the counter value to determine how many iterations (chunks) the inner loop should run. A scalar-valued *current array pointer* is declared here as well, and set initially to the contiguous base array, while the offset is set just as it would be in a standard slice view. For all subsequent iterations of the inner loop, the offset is set to zero, since each supplemental chunk is allocated its own buffer, and the current array pointer is updated to target each successive chunk array. Because the chunks are arranged in a linked list, the local state can only be *incremented*, rather than set arbitrarily, as the current array pointer must advance over all chunks in sequence. When the chunk state is incremented, the current physical vector length is conditionally chosen to be either the chunk length, if more chunks remain to be traversed, or the remainder of the counter modulo the chunk length. Meanwhile, a boolean "spilled?" variable is set every time the local state is initialized to indicate whether the bucket in question has exceeded its initial reservation, so that access to the ancillary pointer arrays can be avoided in the common case.

As described above, however, every access to the chunk array–even those that do not "spill" to secondary chunks–would require accessing the array of counters, which for large chunk arrays would likely cause an extra, unnecessary cache miss. HiRes provides an option to reduce this traffic cost by storing the counter values inline with record cells themselves. In this configuration, an extra "dummy" record is prepended to each bucket's initial reservation in the contiguous base array, enabling the counter and first record(s) to be fetched from memory in a single line if the record size (or its first field group's) is sufficiently small. Presently, only the first field of the array's record type is used to hold the inline counter, and the chunk size in such cases must not be larger than the number of records addressable by an unsigned integer of the same bit width as that field[1]. To accommodate this feature, the chunk array view first checks the inline counter, and only if it is larger than the chunk size does it consult the ancillary counter array, which is guaranteed to have a counter large enough to address the entire dynamically sized bucket.

One problem created by the inline counter format is that meta-arrays with ancillary structures generally re-use (i.e. "copy" by reference) these in their clones: although a clone derived from an initial array might have a new set of underlying record buffers with different contents, the metadata surrounding it, such as histograms, num-valid counters, or masks,

[1] To fix this, it would be necessary to add support for union types to LoRes, which is possible in theory but has not yet been attempted

will remain the same, and so are referenced by a pointer inside the clone. This is trickier when per-bucket counters are stored inline, because the counter array contains accurate values *only* for those buckets that have spilled, and so cannot simply be passed. To handle this quirk, the chunk array's clone method marks derived arrays specially, and their resulting views include a `Mux()` (ternary conditional) expression to arbitrate between the inline counters of the *original array* from which they derive, and the counter array passed by reference. Thus, the data buffers of chunk arrays may stay "live" longer than the operator DAG alone would suggest.

Loop Generation & LoRes Output (Abstraction #3)

The final DAG form of representation between HiRes and LoRes code wraps each node's LoRes operator code in a concrete loop nest derived from the set of cursors that it owns. This LoDag format also explicitly separates operator implementation code from code needed to allocate the node's buffers, which is necessary because these allocations are redistributed between DAG nodes after the latter have themselves been scheduled in a fixed execution order. This lowering comprises three steps: the emission and assembly of array view state declarations at each added level of loop nesting, special management of the so-called "chunk cursor" for chunked arrays, and finally the scheduling and allocation of nodes.

Cursor-based state update insertion

Before loop-nest generation proceeds, a cursor-based analysis pass examines the totality of the LoDag and builds a mapping from each cursor to the set of array view levels tied to it. Then, loop-nests are formed by re-examining each node individually, and choosing one of its views to be the *leader*, and its layers determine the loop nest for the node. In most cases, a node's leftmost (in HiRes syntactic order) input view is elected as the leader, as by convention in HiRes this is the input that gets scanned, and whose valid records drive access to their counterparts in any other possible inputs (in a `Gather()`, for instance, the left input is scanned, causing the indices it names to be accessed in the right input; a join, similarly, probes the right-input hash-table with keys read sequentially from the left). However, if a node lacks any inputs, because it refers to an external input to the HiRes program, or is a "virtual input" referring to a value from outside of a fused node, the node's output view takes the lead instead.

Loop nest layers are then added to the node by examining each layer of the leader view, starting from the outermost, which corresponds to the innermost loop. If the current view's cursor is owned by the node, then a new loop is wrapped around the current LoRes code for that node, and its bounds are set by querying the *maximum cursor* attribute of the leading view. On either side of the new loop, local and global state updates are inserted by querying the global mapping, built in the previous step, of all views that depend on the current cursor. This ensures that all views dependent on said cursor emit state updates, even if they are not even necessarily part of the current node's inputs and outputs (as would be the case for a fused DAG's enclosing node, whose own cursor controls state for the fusion depth of all interior nodes).

Managing the "Chunk Cursor"

Because the chunk meta-array splits into two nested views–and thus loops–the HiRes lowering process must at some point decide when a node owns the *chunk cursor*, and so adds a loop over chunks. Though it would seem obvious to assign the chunk cursor to whichever node owns a cursor that controls an inner (bucket) chunk view, this is insufficient to achieve desirable fusion behavior.

Consider the case of two chained, radix-partitioned hash joins. Here, the initial left-hand (probe) input consists of an array of vectors, where the number of number of vectors is the number of radix partitions, which is also the number of output "buckets" in the resulting chunk arrays of both joins. The enclosing fusion node owns the cursor for this nesting depth, but it must not own the chunk cursor; rather, each internal join node must control a separate chunk cursor, because the number of chunks per bucket can change between them.

Instead, a node owns the chunk cursor if it also owns a cursor that controls:

- A level that subdivides the chunk array
- The vector cursor when the chunk array is the innermost division
- The chunk array *and* the vector cursor

The last condition catches cases where a node contains a chunked array and does not participate in fusion, while the first allows the chunk array's buckets to be decoupled from the chunk lists themselves; the second handles intermediate cases where no level further subdivides the chunk array.

Allocation & Scheduling

The last refinement includes the insertion and placement of buffer allocations, and the arrangement of all nodes in the two-dimensional DAG into a one-dimensional sequence of loops. In the latter case, the order of execution is determined simply by post-order traversal the DAG from the output in left-to-right order; if an alternative order of execution, then this must be achieved by re-ordering the relevant statements in the HiRes input program.

Allocation is slightly more subtle. Many buffers have lengths known *a priori* as a function of the program's input sizes, and so they can be allocated at the start of the compiled function. Ideally, all buffers would be pre-allocated in this way, so that their allocation does not interfere with operator execution by e.g. causing context switches that evict cache and TLB entries.

However, when the `Compact()` operator is used with a histogram computed by `Offsets()`, the size of the resulting output buffer can only be known dynamically, and only after the execution of all operators on which it indirectly depends. Moreover, this dependence propagates upwards through all nodes after the compaction (and where no reduction intervenes), meaning their buffers' allocations cannot be inserted prior to this point either. For this reason, the compiler performs a dependency analysis on the LoRes expression ASTs contained in both the operator codegen result of each node, and its separate block of allocation statements. The analyzer first examines the main body of code in each node of the scheduled DAG, taking note of every LoRes identifier that appears in the root of an assigned l-value inside that block, and marking these as *produced* by that node. Then, it examines the allocation block separately, and notes each identifier that appears in any expression appearing in that code as a dependency.

Because the nodes themselves have already been scheduled, the identifier whose producer is latest in the schedule determines that earliest point at which that node's allocation can be inserted. When a compaction operator is elaborated, it emits a single variable declaration which is assigned to contain the very last entry of the input histogram, and all subsequent allocations are therefore seen as dependent on this symbol. Thus, subsequent nodes' allocations will be hoisted precisely to the point at which compaction was added to the schedule. This ensures that allocation is performed as soon as possible, and in as few places as possible, while still allowing for dynamically sized non-chunked arrays.

5.2 LoRes Compilation Pipeline

The LoRes intermediate format was designed to support both HiRes operator elaboration (codegen) and pre-C++ transformation passes[2]. Since C++ is its ultimate compilation target, its syntax (Figure 5.3) closely resembles that of C, with some simplifications that make it easier to manipulate. It is an imperative language with serial and parallel for-loops, if-statements, functions, arrays, pointers, and bit-wise operations. Its type system (Figure 5.4) is explicit and static, and includes the same set of primitives as HiRes, but adds a non-primitive struct type whose fields can contain pointers and other structs, and through which the meta-arrays and array views of HiRes are realized. Its array values track their lengths implicitly, like C++ `std::vector<T>`s, which simplifies operator codegen somewhat.

The main value of LoRes is that it mainly exists as fully typed ASTs that support type-sensitive transformations. A few additional features facilitate this even further. For loops have fixed termination conditions, as their bounds and strides are computed *once* per execution of the loop, rather than once per iteration, allowing them to be hoisted out. Static single-assignment (SSA) "phi" expressions are baked into the language and can be executed in its interpreter. LoRes ASTs can also replace regular string-valued identifiers with a more efficient `LoSym` symbol that allows transformation passes to perform reference equality comparisons, rather than using string or hash code equality, which is much faster. The same construct also encodes SSA re-numbering, which can later be easily removed.

[2] It originated as the "IRep" intermediate format in our prior work [35]. This thesis, in addition to changing its name, prunes some unnecessary constructs, expands the type system, and describes new transformation passes.

Without reviewing the full details of the LoRes language, the rest of this section presents some of the transformation passes used for both optimization, and for realizing the semantics of the HiRes constructs it was designed to represent.

AST-Based Static Single-Assignment Form

Static Single-Assignment (SSA) form is a well-known compiler transformation pass [2], dating back at least half a century [62], that vastly simplifies other transformation and analysis routines by converting programs to a form of representation in which each variable is assigned to exactly once. The single-assignment property ensures that multiple uses of a single identifier always refer to the same value, which greatly facilitates optimizations such as common-subexpression elimination (Section 5.2), and enhances the effectiveness of dead-code elimination, as updates to an original variable occurring after their last read point become updates instead to new symbols whose values are never used.

For example, in the pseudocode (not LoRes) program above containing only assignment statements, the left-hand side would be rewritten to the right. The SSA variant makes clear that (1.) the assignments to y_1 and x_2 are never used, and so candidates for elimination, and (2.) the expression $x + 1$ carries the same meaning ($x_0 + 1$) both times that it occurs in the original code, and so is a real common-subexpression.

Handling Control Flow in AST-SSA

SSA conversion is less straightforward in the presence of statements that alter control flow, and this is where the LoRes SSA format begins to differ from more conventional implementations. Normally, the SSA transformation is applied to a control flow graph (CFG) program representation in which each node comprises a basic block of operations with one entry point and one exit. In such cases, *join nodes* n_3, reachable by two distinct paths from different source nodes n_1 and n_2 in which a single original variable x is assigned, require the insertion of *phi nodes* ($x_3 \leftarrow \phi(x_1, x_2)$) that rename x to indicate the two possible input values that x may have at n_3.

However, LoRes does not encode any explicit CFG, as it strictly a scoped AST representation. For example, consider the (pretty-printed) LoRes program in Listing 5.1. A mere parse of its code yields the AST of Figure 5.7a. In contrast to the CFG of Figure 5.7b, the AST's edges alone do not supply sufficient information about the control relationships in Listing 5.1: for example, `For` node (5) in Figure 5.7a does dominate the assignment in node (9), but the semantics of the `For` construct clearly imply a path $n_5 \xrightarrow{+} n_9$ in the CFG; similarly, while node (5) appears on only two edges in the AST, it's CFG homologue has two incoming and two outgoing edges. In general, a CFG node n_i from which a path $n_i \xrightarrow{+} n_j$ exists to another node n_j is said to *precede* n_j, which is its *successor*; if *every* path to n_j includes n_i, then n_i *dominates* n_j, according to common parlance [10].

Clearly, these differences between AST and CFG structure have implications for ϕ-node insertion: any nodes with multiple inputs in the CFG might necessitate a ϕ, even if this

stmt	::=	Dec(*id*), DecAssign(*id*, *expr*)	l-value statements
		Assign(*lval*, *expr*)	
		Alloc(*lval*, [*length*]), Free(*lval*)	
		AssignReturn(*lval*, App(...))	
		PrefixSum(*lval*), RotRight(*lval*, *n*)	“Vector” Operations
		Block(*stmt* ∗)	Control Flow
		IfElse(*expr*, *stmt*, *stmt*)	
		Return(*expr*)	
		ForPar(index=, min=, max=, stride=, *stmt*)	
		FuncDec(*id*, (*id*, *prim*)∗, *stmt*, [*prim*])	Functions
		App(*id*, [*expr*∗])	Call/“Apply”
		Printf(fmt=*str*, [*expr*∗])	Misc. / Output
		UseSym(*sym*, *sym*)	
expr	::=	*lval*	
		Plus(), Minus(), Mul(), Mod(), Div()	Numerics (*expr*, *expr*)
		Neg(), Pow(), Pow2(), Log2Up()	
		ShiftLeft(), ShiftRight(), BitAnd()	Bitwise ops.
		BitRange(*expr*, *expr*, *expr*)	
		Const(*num*), True, False, Null	Constants
		FloatConst(*float*), DoubleConst(*double*)	
		Mux(*expr*, *expr*, *expr*), Phi(*expr*, *expr*, *expr*)	Booleans
		And(), Or(), Not()	
		Greater(), GreaterEq(), Equal()	
		Less(), LessEq()	
		NumEntries(*expr*), Size(*expr*), Cast(*expr*, *type*)	Misc.
lval	::=	Id(*str*), LoSym(*obj*, [*num*])	Symbol-Like
		Ref(*lval*), Deref(*lval*), Subsc(*lval*, *expr*)	Pointers & Arrays
		Field(*lval*, *id*), UField(*lval*, *num*)	Structs

Figure 5.3: (Sugar-Free) Syntax of the LoRes Language (Summary)

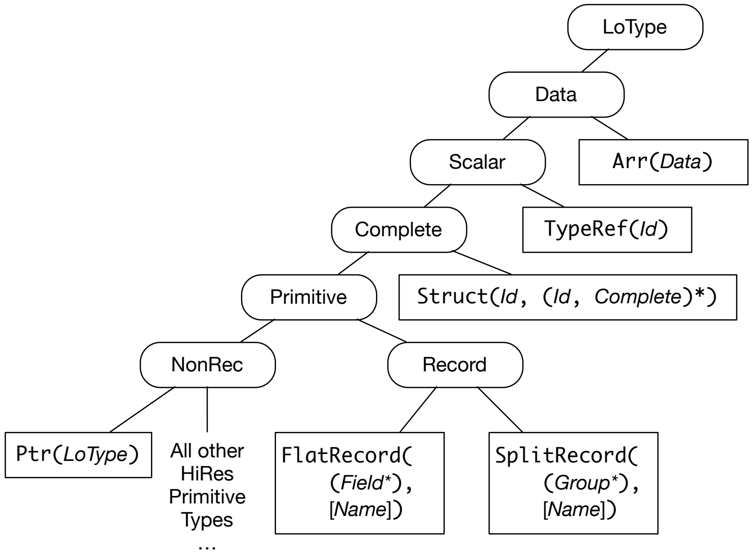

LoType
Data
Scalar
Arr(*Data*)
Complete
TypeRef(*Id*)
Primitive
Struct(*Id*, (*Id*, *Complete*)*)
NonRec
Record
Ptr(*LoType*)
All other HiRes Primitive Types ...
FlatRecord((*Field**), [*Name*])
SplitRecord((*Group**), [*Name*])

Figure 5.4: The LoRes Type Hierarchy

$$\begin{array}{l} x \leftarrow 1 \\ y \leftarrow x + 1 \\ y \leftarrow x - 1 \\ y \leftarrow y + x + 1 \\ x \leftarrow y \\ \textbf{return } y \end{array} \qquad \begin{array}{l} x_0 \leftarrow 1 \\ y_0 \leftarrow x_0 + 1 \\ y_1 \leftarrow x_0 - 1 \\ y_2 \leftarrow y_1 + x_0 + 1 \\ x_2 \leftarrow y_1 \\ \textbf{return } y_2 \end{array}$$

(a) Original Program ⇒ (b) SSA Form

is not obvious from the AST. However, rather than building an explicit control flow graph, the LoRes compiler walks the AST in depth-first, post-order traversal, and inserts ϕ's by the following rules at each node n:

1. If n has any children, replace them according to these rules

2. If n is an `If` with no else-clause, find all symbols x updated within n's children, with previous name x_{i-1} and new name x_i, and whose original scope of declaration dominates n. Rename them after node n with a phi: $x_{i+1} \leftarrow \phi(x_{i-1}, x_i)$

3. If n is an `IfElse`, then apply the above rule but divide the updated symbols into X_{if}, X_{else}, and X_{both}, according to which of the if-else bodies modify them. For each $x_i \in X_{if}$, append a phi-assignment $x_{i+1} \leftarrow \phi(x_{i-1}, x_i)$ selecting between its prior value and the value in the if-clause; for $x_j \in X_{else}$, select $x_{j+1} \leftarrow \phi(x_{j-1}, x_j)$ again between the pre-if value and its else-clause update. However, for $x_k \in X_{both}$, select $x_{k+1} \leftarrow \phi(x_{if}, x_{else})$, where x_{if} and x_{else} represent the most recent renames of x inside the if- and else-clauses, respectively.

```
var x: Index := 0











For(i = 0...100 by 1) {

    x := x+i
    x := x-1
    For(z = 0...10 by 1) {

      x := x+z
    }

    If (x>i) {
      x := x+1
      Nop
    }


}



var y: Index := x*x
```

Listing (5.1) Ex. LoRes program

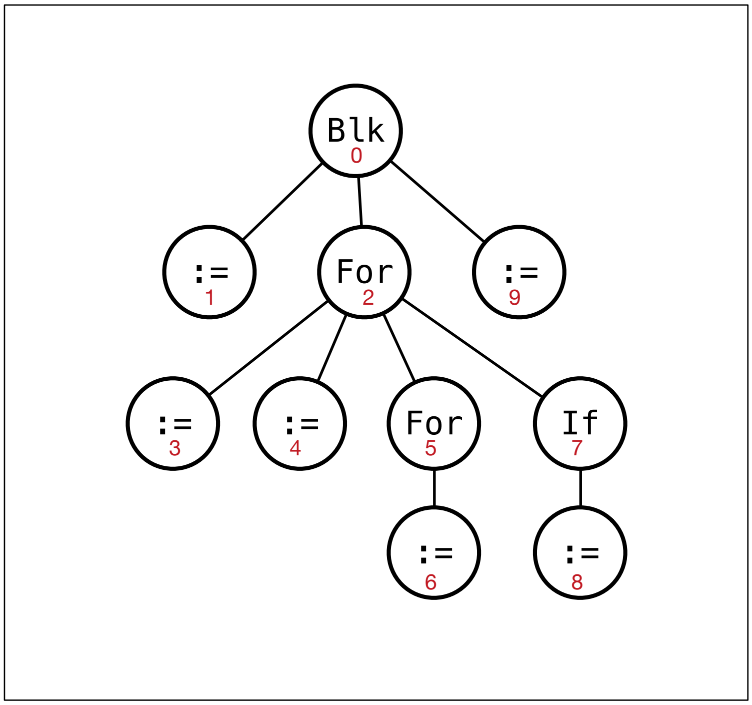
```
var x0: Index
var x1: Index
var x2: Index
var x3: Index
var x4: Index
var x5: Index
var x6: Index
var x7: Index
var x8: Index
var x9: Index
var x10: Index
x0 := 0
For(i = 0...100 by 1) {
    x1 := Phi(i==0,FakeUse(x0),FakeUse(x8))
    x2 := x1+i
    x3 := x2-1
    For(z = 0...10 by 1) {
        x4 := Phi(z==0,FakeUse(x3),FakeUse(x5))
        x5 := x4+z
    }
    x6 := Phi(10>0,FakeUse(x3),FakeUse(x5))
    If (x6>i) {
        x7 := x6+1
        Nop
    }
    var cond: bool := x6>i
    x8 := Phi(FakeUse(cond),x7,x6)
}
x9 := Phi(100>0,FakeUse(x0),FakeUse(x8))
var y0: Index
var y1: Index
y0 := x9*x9
```

Listing (5.2) SSA-Transform of Listing 5.1

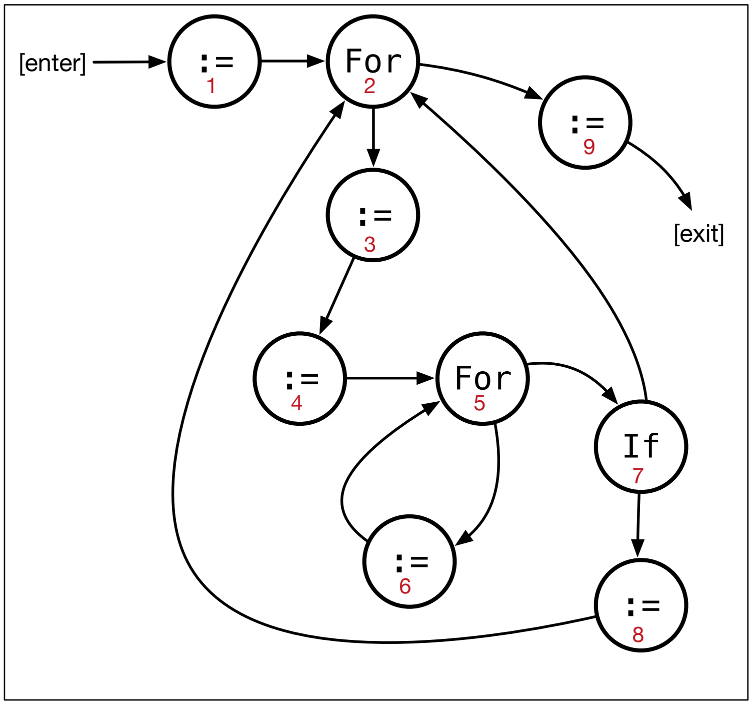
4. If n is a for loop `For(i=0...n) { }`: find all symbols x modified by n and declared outside its scope. Insert a phi-assignment $x_{i+1} \leftarrow \phi(x_{i-1}, x_i)$ *inside* the loop body at its beginning, and *replace* all uses of x_{i-1} inside the loop body with x_{i+1}. Append phi-assignments $x_{i+2} \leftarrow \phi(x_{i-1}, x_i)$ just *after* the whole for loop n. [3]

The above rules work because only loops and if-else statements can introduce additional edges into the CFG beyond those implied between successive statements within a block. More precisely, a loop node in the AST implies an extra control edge from its *rightmost,*

[3] Note that in LoRes the loop bound and stride expressions are calculated exactly once, so any symbol that appears inside one will *not* see any updates made to it by statements inside the loop body. Supporting while loops would require renaming symbols with ϕ-expressions inside the termination condition, which the compiler presently does not.



(a) AST For Listing 5.1



(b) CFG For AST in Figure 5.7a

deepest child (e.g. node 8 for the for loop node 2 in Figure 5.7a) back to itself in the CFG, while an if node in the AST implies an extra control edge from its rightmost child to the if-statement's successor (an if-else node adds an edge from the rightmost, deepest child of the if body; the else body's equivalent already has such an edge).

Thus, the only way a CFG node can acquire multiple incoming edges is if it is a loop node, or is the immediate successor of an if node. In the loop node case, insertion rules guarantee a ϕ-node gets added just after the loop node in the CFG and prior to its children, thereby capturing for them the convergence of the loop's two incoming control paths, while the ϕ-node added after the loop in the AST captures the same convergence of paths for non-child nodes to which a path from the loop exists in the CFG. Meanwhile, the insertion of ϕ-nodes as immediate successors of if-statements in the AST guarantees that such nodes cover the two converging paths to from the if node to its successors in the CFG. These ϕ-nodes might not strictly dominate all AST-successors in the CFG, as a successor may have other incoming edges, but by the foregoing logic, the successor must then itself be either a loop node or the successor of another if-else, in which case the requisite ϕ-nodes would also have been inserted.

Therefore, *any node in the original, implicit CFG that joins multiple paths from distinct sources assigning a symbol x will be strictly dominated by a ϕ-node for x in the CFG implied by the post-SSA AST.* This set of ϕ-nodes is sufficient, but possibly (much) more than necessary, as shown by Cytron et al. [10]. Since performance of the SSA transformation and its dependents has not yet proven to be a bottleneck, the pruning of superfluous ϕ-nodes has not been attempted (and would likely entail constructing a representation of the implicit CFG explicitly in order to make use of known algorithms).

LoRes L-Values & SSA

Assignment statements may contain l-values that are more complex than a simple variable name. Whenever an array subscript, pointer dereference, or structure field is modified, the symbol *at the root* of the assigned l-value is renamed to ensure that any subsequent uses observe the result of the assignment, even if they access it by way of a different l-value expression. This is a conservative analysis: if two array subscripts `a[i]` and `a[i+1]` differ, they are treated as if they refer to the same memory, since it is impossible to determine in the more general case whether `a[i]` and `a[j]` are aliases. Furthermore, whenever the a pointer-typed value l_1 is derived from some l-value l_2, its own l-value's root must be marked as a possible alias for the root of l_2, and vice versa. Whenever a symbol is renamed, so too must be all symbols in its alias set. The example array code in Listing 5.3 and its corresponding SSA form in Listing 5.4 show how the preceding rule applies.

LoRes AST Additions for SSA

The concrete SSA format of Listing 5.4 encodes several seemingly superfluous details. These are mostly motivated by the desire to make the SSA-form AST executable in the LoRes interpreter–

```
var a: ptr_array__Index








Alloc [ a, 10 ]


Deref(a)[0] := 1
For(i = 1...10 by 1) {



    Deref(a)[i] := Deref(a)[i-1]


    Deref(a)[5] := i


    Deref(a)[i] := (Deref(a)[i])+1
}
```

Listing (5.3) A LoRes Array Loop

```
var a0: ptr_array__Index
var a1: ptr_array__Index
var a2: ptr_array__Index
var a3: ptr_array__Index
var a4: ptr_array__Index
var a5: ptr_array__Index
var a6: ptr_array__Index
var a7: ptr_array__Index
Alloc [ a0, 10 ]
a1 := a0
UseSym(a0,a1)
Deref(a1)[0] := 1
For(i = 1...10 by 1) {
    a2 := Phi(i==1,a1,a5)
    a3 := a2
    UseSym(a2,a3)
    Deref(a3)[i] := Deref(a2)[i-1]
    a4 := a3
    UseSym(a3,a4)
    Deref(a4)[5] := i
    a5 := a4
    UseSym(a4,a5)
    Deref(a5)[i] := (Deref(a4)[i])+1
}
a6 := Phi(10>1,a1,a5)
```

Listing (5.4) SSA-Transform of Listing 5.3

which has been of significant aid in debugging the compiler and the relevant transformation passes–while others are the result of subtle interactions between said transformation passes.

The ϕ-node construct is realized in LoRes IR as the Phi($cond, e_1, e_2$) expression, which explicitly embeds a boolean-valued *path condition* as its first input, providing enough information for the runtime to decide which side of the Phi()'s two input values should be chosen as the next value of the renamed variable. In general, such information would not be needed, as ϕ-nodes exist only temporarily inside the compiler to track flows of information between subsequent uses of a single identifier so that program semantics may be preserved under various transformations. The LoRes compiler also removes all ϕ-nodes and collapses subscripted variable names $x_0 \ldots x_n$ back to x after all passes that require SSA form have been completed, but as these passes are often difficult to implement correctly, the ability to execute their SSA-formatted outputs directly has proven highly useful.

When array pointers are renamed after l-value assignments in Listing 5.4, the new name is assigned to the old with an explicit assignment statement, ensuring that the SSA version remains executable. An additional UseSym(old, new) statement is also inserted to create a backwards dataflow relationship, indicating that the old pointer "sees" the new one, and thus also any updates to one of its derived l-values, so that this relationship is clear to any later dead-code elimination (DCE) passes[4].

AST-Based Common Subexpression Elimination

A chief purpose of the foregoing SSA conversion algorithm is to enable common-subexpression elimination (CSE), which identifies some cases in which two or more expressions are known to compute the same value, and eliminates the redundant computation by assigning that value once to a new temporary variable to be used multiple times. SSA format make CSE possible by ensuring that a single identifier only ever refers to one particular value however many times it is used inside the program, so any two occurrences of an expression such as `a1+b2+1` are guaranteed to have the same meaning. Otherwise, CSE would require almost intractable safeguards to ensure that expression renaming is reset whenever a new value is assigned to a variable, or when a given program location might be reachable via multiple control paths that each update the same variable (the ϕ-node insertion condition). SSA solves these problems. The utility of CSE in Ressort is not so much to reduce arithmetic computation in HiRes operator implementations as it is to clean up the litter strewn by the code generators that support the language's underlying datatypes, and to facilitate analysis passes that require identifying when two pairs of array bounds are identical, among other things.

Section 5.1 describes the source of LoRes code bloat in terms of array view state generation. In order to keep the architecture of the HiRes compiler's frontend as simple as possible, it was decided that the internal API of each array view type should simply supply methods to re-calculate whatever state is required and assign it to a new temporary variable. Instead, each view simply supplies its own independent offset and bounds calculations, and relies on the CSE and DCE passes in the backend to coalesce any resulting proliferation of state updates into the minimal set of required declaration and assignment statements. This, in turn, will make it much easier to identify when, for example, multiple access to a single buffer necessarily refer to the same location in memory, as opposed to locations separated by significant distance, thereby enhancing the efficacy of analyses like the communication cost model of Chapter 7. Listing 5.5 shows how easily redundant size and offset calculations amass even even for simple HiRes programs like parallel scan operator introduced in Listing 3.1 of Chapter 3, while 5.6 shows the effectiveness of CSE-based *post-hoc* tidying.

Challenges of AST-CSE

The benefit of a CSE pass is to remove the complexity of redundancy elimination from the front end of the compiler, but it comes at the cost of a complex CSE pass implementation.

[4] The LoRes language specifies that pointers may not alias. Since the SSA format clearly violates this requirement, the `UseSym()` was added as a way of marking aliasing explicitly. Such a relationship could likely be inferred by the dataflow analysis pass if it made us of the type-checked AST to ascertain when assignments were references, but it does not do so presently.

```
var t_slen: ptr_array__Index
const narr0: const_Index := 16
var avg_len0: Index := 16/narr0
var rem0: Index := 16%narr0
var idx: Index := 0
// [...]
const narr3: const_Index := 16
var avg_len3: Index :=
    (NumEntries(Deref(table->group_0)))/narr3
var rem3: Index :=
    (NumEntries(Deref(table->group_0)))%narr3
const narr4: const_Index := 16
var avg_len4: Index := 16/narr4
var rem4: Index := 16%narr4
const narr5: const_Index := 16
// [...]
const narr15: const_Index := 16
ForPar(fcur0 = 0...narr15 by 1) {
    const off0: const_Index := Mux(fcur0>0,
      (avg_len3*fcur0)+rem3,avg_len3*fcur0)
    const off1: const_Index := Mux(fcur0>0,
      (avg_len4*fcur0)+rem4,avg_len4*fcur0)
    const off2: const_Index := Mux(fcur0>0,
      (avg_len1*fcur0)+rem1,avg_len1*fcur0)
    var pvl: Index := Mux(fcur0>0,
      avg_len5,avg_len5+rem5)
    // [...]
```

Listing (5.5) Initial LoRes for Listing 3.1

```
var t1: Index
var t2: Index
var t3: Index
t1 [:=] NumEntries(
  Deref(table->group_0))
t2 [:=] t1/16
t3 [:=] t1%16
var t_slen: ptr_array__Index
ForPar(fcur0 = 0...16 by 1) {
    var t6: Index
    var t7: bool
    var t8: Index
    var t10: Index
    var t12: Index
    var t13: Index
    var t20: Index
    var t21: Index
    t6 [:=] fcur0
    t7 [:=] t6>0
    t8 [:=] t2*t6
    t10 [:=] Mux(t7,t8+t3,t8)
    t12 [:=] Mux(t7,t2,t2+t3)
    t13 [:=] Safe(t6)
    var ocur: Index
    ocur := 0

    // [...]
```

Listing (5.6) CSE-Transform of 5.6

CSE is challenging to implement in LoRes because it is performed on block-scoped ASTs, rather than at the basic-block level of representation, as would normally be the case.

Ideally, CSE could be implemented with a simple algorithm: traverse the SSA-format CFG (in any order), examine the expressions encountered in each basic block, and look each one up in a hash-table, and replace it with its renamed temporary variable if it has been seen before, or create a new one if it has not. Complex expression ASTs containing more that a single two- or three-input operation should be broken down into their smallest atomic units, starting from the leaves of the tree, each of which is renamed separately. Then, all new temporary variables could be declared globally, and their values can be assigned at the earliest point where both of their inputs have been assigned.

However, the block-scoped AST context thwarts this simple CSE algorithm in two ways. First, any new temporaries cannot simply be introduced as global variables, and must have a precisely defined scope to preserve the original semantics. One reason for this is parallel loops: if an expression uses a variable with scope local to a parallel loop, then its renamed temporary *cannot* be defined with scope higher than the parallel loop body, or else it would not end up with *thread-local* storage. For example, in Listing 5.6, the variable `t7`–which has the value `false` for the 0th iteration and `true` for all others–must not be declared above the loop. This would violate program semantics and would cause a data race as multiple threads (iterations of the parallel loop) attempt to assign different values to a single memory location. Temporaries must also not be declared at too "low" a scope, such as at the site of their eventual assignment, as they must be visible to all AST locations from which all their inputs are accessible. Thus, the declaration of a temporary variable should always occur at precisely the point of declaration of the input whose scope is dominated by the scope of every other input. Such an input must exist, as otherwise the original expression could not have seen two inputs that have no domination relationship.

Secondly, care must be taken to ensure that expressions with side-effects, such as array subscripts that may cause a memory protection fault if out of bounds, do not get hoisted outside of enclosing if statements that control whether or not they should be evaluated. When an expression containing an array subscript or pointer dereference is renamed, the resulting temporary's declaration scope is limited to the block in which the original expression was used. Furthermore, expressions with a pointer or array *type* are never renamed, so CSE will not create any new aliases.

A temporary's assignment may occur far away from its point of declaration. This is the case for instance when one of the temporary's inputs is only assigned inside an if-else statement, but is visible over a much wider scope. The LoRes CSE pass sets the value of temporaries at the point of assignment of their last input dependence in program order (i.e. depth-first AST traversal). If a temporary's last dependency gets assigned inside of an if statement, then the temporary will also be assigned there conditionally; if the last dependency is originally (before SSA and CSE) assigned inside a for loop, then the SSA conversion will have renamed that dependency and appended a ϕ-node after the loop on which the temporary now depends instead, meaning that it will be assigned just *after* the loop.

Passes to Support HiRes

Some LoRestransformation passes are necessary to implement the semantics of HiResoperators fully, while others merely improve the resulting generated code. All rely heavily on the full static typing of LoResASTs, which motivated its introduction as an IR in place of C or C++.

Record Group Elimination

The record-group-elimination pass applies whenever a HiResrecord type is instantiated in a LoResarray, and acts as an array-of-structs to struct-of-arrays transformation. Recall that the split record type (Chapter 3, Section 3.2) requires each *field group* to be stored in a separate LoRes/C++ array in order to implement the semantics of the `Zip()` and `Project()` operators. Record-group elimination converts any split record types into a combination of struct types for each field group, and updates all expressions derived split record-typed values accordingly. Afterwards, all other flat record types are converted to structs as well, so they are suitable for direct mapping to C++.

Struct Scalarization

When temporary scalar values are declared for struct types derived from HiResrecords, struct scalarization splits them up into separate temporary declarations for each primitive-valued field. This enables DCE to prune unused fields and, consequently, accesses to their originating arrays.

Unused Pointer Removal

Pointer-typed variables cannot participate in DCE prior to SSA collapse, due to conservative alias analysis. Since chunk-arrays' views generate potentially a large number of unnecessary pointer state, a final "unused pointer removal" pass is applied after all SSA/CSE/DCE passes and conversion back to non-SSA form, when `UseSym()` statements have been removed. This pass simply discards pointer-typed variables that never appear in an RHS expression.

Common-Predicate Merge

Since execution masks can change at the boundary between any two operators, they are implemented in the HiRescompiler by wrapping each operator-specific code generation in a LoRes`if` statement with whatever mask value is present in its leader input. This results in a sequence of separate `if` statements even when operators are fused at the vector level. To fix this, a LoRes-level predicate merge pass discovers adjacent `if`-blocks with syntactically identical condition clauses. This is not correct in the general case, since an the body of such a statement could mutate values used in the condition, but currently no HiRescode generator does this.

Chapter 6

Evaluation

Although the goal of Ressort was to enable communication-reducing query plan transformations that should shape the design of database-processing platforms that do not yet exist, this chapter nonetheless presents preliminary empirical measurements from server-class processors similar to those widely deployed in datacenters today. Machine-level plan decisions do matter on existing hardware, and the observed variation between them indicates the difficulty of avoiding performance pitfalls, even if these do not always correspond directly to differences in communication requirements, and instead result from other pathologies.

While the communication-based discussion of Chapter 2 and analysis of Chapter 7 can be said to view the world in terms analogous to a physicist's hypothetical frictionless surfaces and perfectly elastic collisions, the measurements below confront the real bottlenecks of non-ideal systems. Significant compiler development effort was required to support features that resolve such quirks of general-purpose processors, whether or not they alter traffic.

6.1 Experimental Platform

Hardware

Results were measured on two Intel Xeon E5-2667 parts of different generations (launched in 2013 and 2016), representing slightly different microarchitectures, as shown in Table 6.1. Both systems are dual-socket and have two NUMA domains, with 8 cores (16 hyperthreads) per socket.

Name	Model	Clock	Sockets	Cores	Caches	Mem. BW	TLBs
Ivy Bridge	E5-2667 v2	3.30GHz	2	16	32K, 256K, 25M	59.7GB/s	64, 512 ent.
Broadwell	E5-2667 v4	3.20GHz	2	16	32K, 256K, 25M	76.8GB/s	64, 1536 ent.

Table 6.1: Intel Xeon Server Platform Specifications

lineitem (SF * 6M records)			part (SF * 200K records)	
Field	**Type**		**Field**	**Type**
l_extendedprice	LoFloat		p_partkey	UInt32
l_discount	LoFloat		p_container	UInt8
l_tax	LoFloat		p_brand	UInt8
l_partkey	UInt32		p_size	UInt8
l_quantity	UInt32		p_type	UInt8
l_shipmode	UInt8		p_retailprice	LoFloat
l_shipinstruct	UInt8			
l_shipdate	UInt16			
l_receiptdate	UInt16			
l_commitdate	UInt16			
l_returnflag	UInt8			
l_linestatus	UInt8			

Table 6.2: Encoding of `lineitem` and `part` Relations

Software

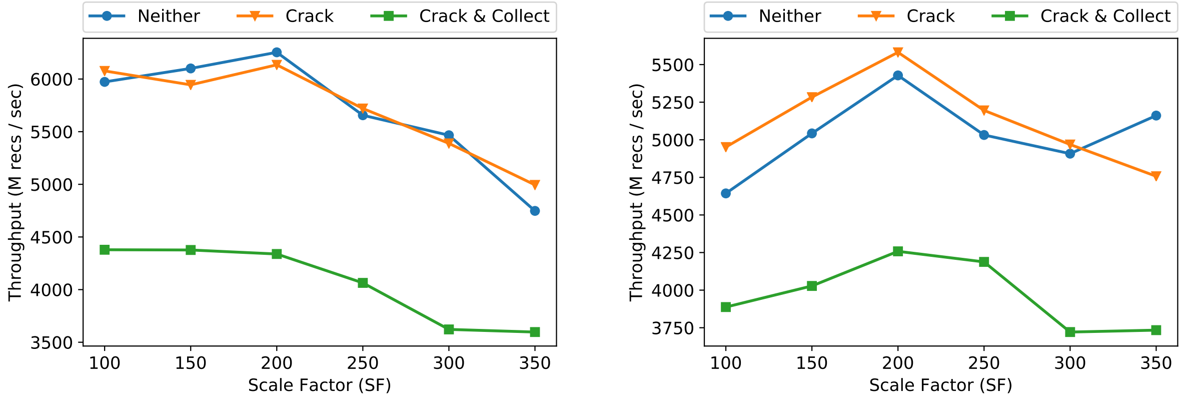
For the purposes of evaluation, the Ressort compiler stack was used to generate C++ code compiled with GCC [14] version 7.2 or higher using the `-O3` level of optimization and with OpenMP support enabled. The resulting binaries were executed on Ubuntu Linux version 16.04 with the `OMP_PROC_BIND` flag set.

6.2 "TPC-H"-Like Queries

All results were measured in a simulated column store environment, similar to that of Schuh et al. [59], wherein custom C++ code generates columns of uniform random attributes as part of a schema closely resembling that of the TPC-H benchmark [70], *but which does not conform to the TPC-H specification.* In particular, all string-valued attributes are dictionary-compressed to the smallest byte-aligned integer type large enough to contain their full domain, and any attributes that cannot be compressed this way (such as `l_comment`) are omitted, limiting the subset of queries that can presently be encoded. Table 6.2 shows the encoding for each attribute exactly. The length of each relation is a multiple of the *scaling factor* (SF), and shown in Table 6.2. It is furthermore assumed at the start of query execution that each attribute resides in a separate, contiguous column of memory (column-major order).

Results are presented in terms of throughput, calculated as the number of records processed divided by the runtime. The number of records is generally given by the size of the `lineitem` table, which dwarfs the size of the `part` table by a factor of thirty. Only *execution time* is counted as part of the runtime, and compilation time–in either the Ressort compiler or in GCC–is excluded from consideration.

Furthermore, since Ressort currently lacks a proper DBMS front end, it cannot claim to evaluate "TPC-H" queries since there is no way to input them in SQL form, and no



(a) Ivy Bridge

(b) Broadwell

Figure 6.1: Problem Scaling of TPC-H Q06 (SF=350)

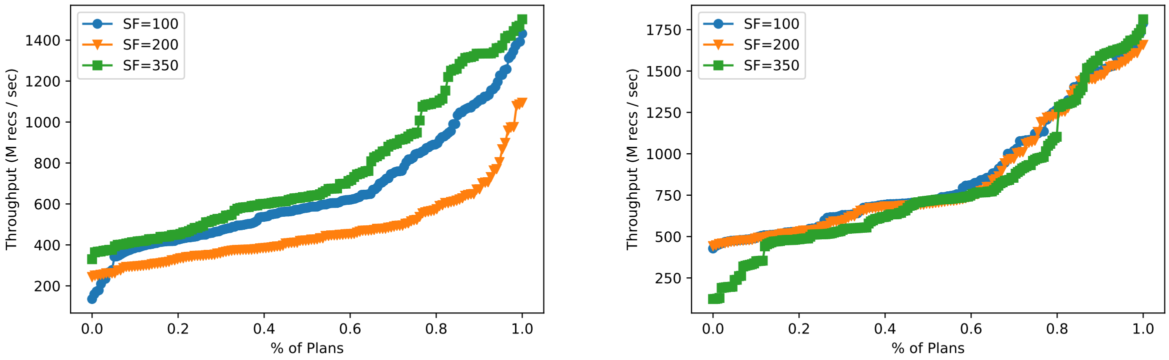
automated query planning or optimization is performed. All measurements below rely on manual implementation of HiRes plan generators in Scala, using the templates described in Chapter 3 Section 3.3. Any logical query optimization, such as ordering the application of selection predicates are applied, or the factoring out of sub-queries, is performed strictly by hand.

Q06

Query 06 (whose SQL is shown in Listing 1.1) consists of selection based on three attributes, followed by an aggregation (sum). The only machine-level plan choice of interest is whether to use predicate cracking, and then whether to collect cracked results between predicates. Figure 6.1 shows the performance of such plans for Q06 as the scaling factor (SF) is varied from 10 to 350. The maximal throughput achieved is ≈5.5 billion records / second, corresponding roughly to 70% of the Broadwell machine's theoretical peak for the cracked plans. The apparently un-cracked curve ("neither") exhibits similar performance, even though this should be slightly in excess of the theoretical peak. The reason for this, as revealed by disassembly of the resulting object files, is that GCC's optimization passes moved loads of the `l_discount` and `l_quantity` attributes after conditional branches, effectively "cracking" the plan under the hood. The proximity of these results to the bandwidth-bound peak is unsurprising, given the simplicity of this query's inner loop, and so they serve as a calibration of the test platforms' bandwidth.

Q19

Query 19 comprises a significantly richer space of machine-level planning choices than Q06, as depicted by the plan generator template in Figure 3.9. Figure 6.2 shows the performance of many plans for Q19 at different scale factors for both of the machines evaluated in this



(a) Ivy Bridge

(b) Broadwell

Figure 6.2: Performance Spectrum of Q19 Plans on Two Machines

study. These *performance spectra* depict each unique plan as a single point on the graph, and plans are sorted from left to right in increasing order of their throughput. The x-axis therefore represents the overall fraction of plans considered in each curve up to each point.

At scale factor SF=350 (the largest considered due to memory constraints), the performance of plans varies by at least an order of magnitude, though the worst of these correspond to degenerate settings of continuously-varying parameters, such as the number of radix bits and size of hash-table buckets. On Broadwell–the more powerful of the two platforms–the peak throughput attained for Q19 is approximately 1.8 billion records per second. Based on a communication cost analysis (Chapter 7) a fully-packed, fully-partitioned plan at SF=350 should be able to run at a rate of 3.95 billion records per second, meaning that the best plans reached 45% of peak performance.

To better understand the impact of planning choices, Figure 6.3 breaks down the SF=350 results into separate curves based on the partitioning strategy employed. At this scale, all three broad classes–non-partitioned ("nopa"), partition-all-attributes ("part-all"), and partition-single-attribute ("part-single")–achieve performance within 33% of the best on each machine, though on both platforms the winning plan uses full partitioning. On the Ivy Bridge machine, the "nopa" curve is noticeably sparse in comparison to the others. This is because many configurations of that strategy have degenerate performance (≈10M recs/sec or worse) that would have taken too long to evaluate and contributed little insight. That such an effect does not manifest on the Broadwell machine is likely due to the fact that the latter has a significantly larger second-level TLB, and so does not thrash as easily when the hash-table (and probe working set more generally) grows beyond its capacity. Otherwise, the two machines' respective plan spectra and peak throughputs tend to resemble each other, as would be expected from two platforms that differ only slightly in terms of out-of-order resources and memory bandwidth.

One structural parameter within the space of partitioned plans is the choice to perform "thread-local" partitioning: that is, instead of imposing a barrier after the histogram

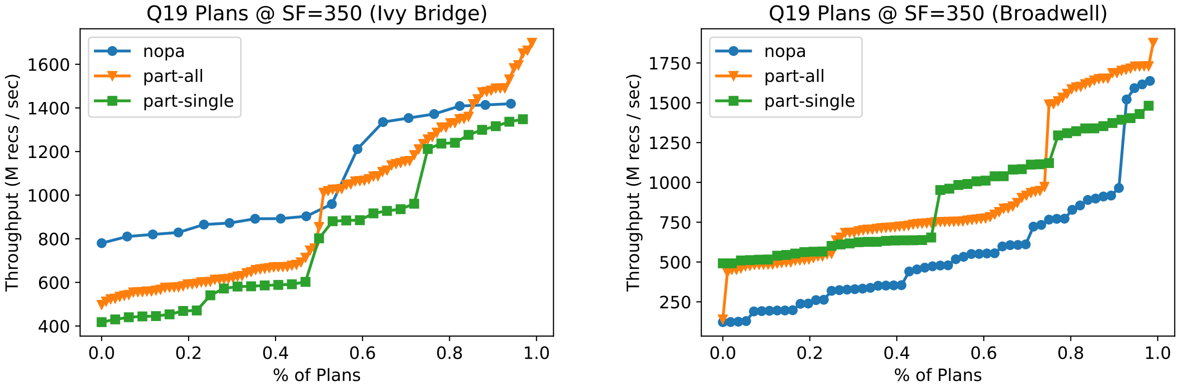


Figure 6.3: Performance Spectrum of Q19 Plans on Two Machines

build phase, and computing global offsets for each thread's partitions into a shared output buffer, each thread moves records to a local output buffer, from which it then performs the probe phase of the join independently. This possibility is exploited more easily in full-query compilation because, as in this case, the final aggregation means that the ordering of the partitioned records does not matter, and no stability is required [1]. Figure 6.4a shows that the impact of local partitioning is only apparent in the "part-single" plans on Broadwell (a similar effect was observed on the Ivy Bridge machine), in which case thread-local partitioning does markedly improve throughput, though the best of these plans still does not perform as well as the best of the fully-partitioned ones.

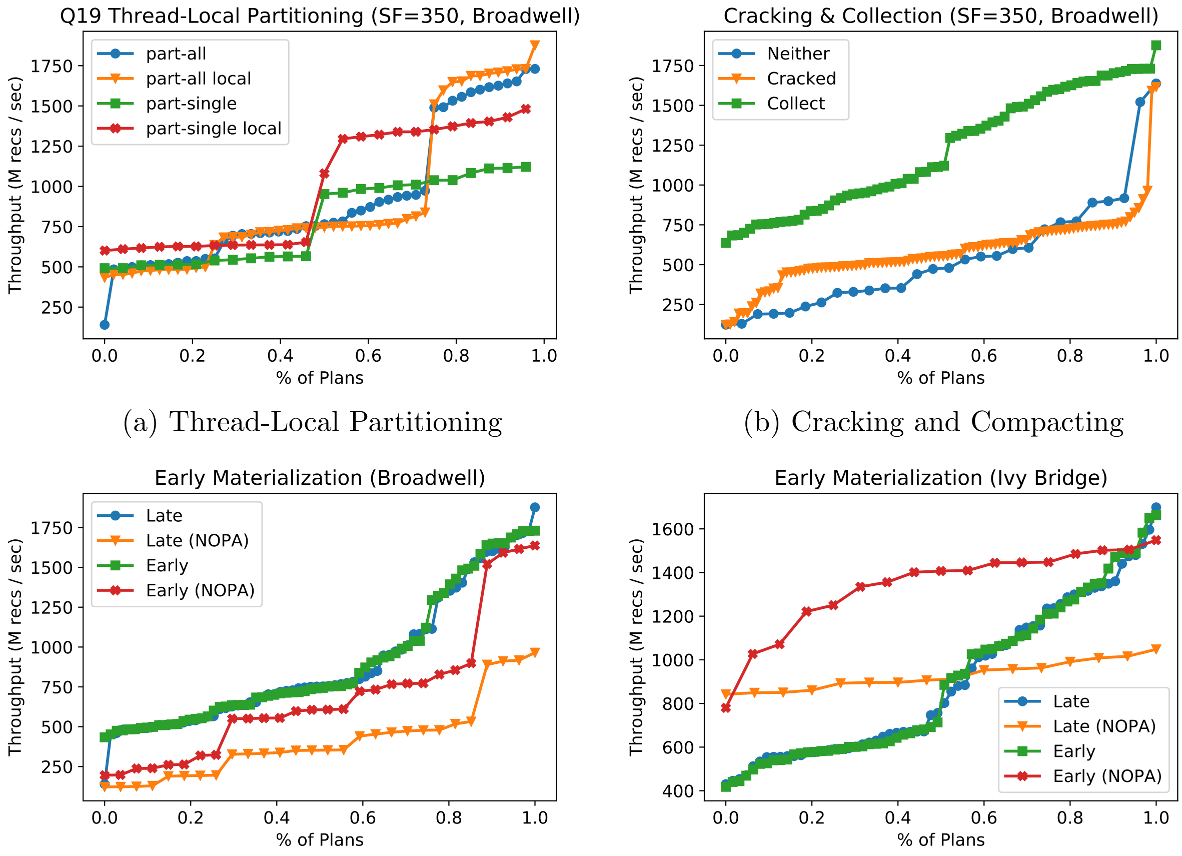
In both cases, many choices beyond partition strategy contribute to significant performance variation. The initial filter of Q19 has a selectivity of 0.08, which at first glance seems small enough to reduce the amount of traffic consumed by scans of additional `lineitem` attributes if cracking were employed, but Figure 6.4b shows that the impact of doing so is negligible. Even if the compiler did not transparently eliminate the "un-cracked" version, the results would still differ little, however. A simple calculation using Pirk's [45] extension to the communication model of Manegold [38] reveals why: if $\sigma = 0.083$ is the likelihood a 4-byte element needs to be scanned, then the likelihood of accessing a particular 64-byte cache line in the array containing it is:

$$1 - (1 - \sigma)^{(64/4)} = 75\%$$

Thus, the communication savings from cracking will be no more than about 25%, but at such a high effective selectivity of cache lines, hardware prefetching is likely to mask any theoretical savings.

In fact, it seems optimal to do just the opposite of cracking: the "collect" curve (green boxes) contains all plans that eagerly load *all* required attributes (for either side of the join),

[1] In theory, this should result in less random cross-socket traffic, though in practice, the plans evaluated did not use a disjointed split (Chapter 3 Section 3.2) operator, so the resulting output buffers may all have been allocated in memory assigned to a single NUMA domain



(a) Thread-Local Partitioning

(b) Cracking and Compacting

(c) Early Materialization (Broadwell)

(d) Early Materialization (Ivy Bridge)

Figure 6.4: Impact of Plan Choices Aside from Pure Partitioning

and coalesce those surviving the initial filter predicates (using the HiRes `Collect()` operator, as supplied automatically by the `Filter` operator macro) into a (fully-packed) buffer of valid tuples before using any of these in partitioning or otherwise[2]. Finally, the early materialization option of Figures 6.4c and 6.4d indicates a marked improvement for non-partitioned plans because the inclusion of post-join attributes into the hash-table eliminates unnecessary cache misses, while for partitioned plans these are already eliminated by partitioning itself.

[2] This improved performance for partitioned plans because, in the experiments presented above, the default fusion choice in partitioning cause a materialization of non-compacted, materialized tuples, as well as a mask, which the foregoing selectivity calculation showed to be nearly an order of magnitude larger than the compacted version.

Chapter 7

Communication-Aware Query Cost Models

Calculating the arithmetic intensity of TPC-H Q06 was straightforward in Chapter 1 because it consists merely of scanning a few arrays sequentially, making its communication requirements easy to compute. It is also possible to adduce the traffic costs of more complex operations, such as joins, and even whole queries that include them, but they require more sophisticated techniques than back-of-the-envelope analysis. Seminal work by Manegold et al. [38] provides a basis for doing so.

They encode common query operations in an algebra of memory access patterns whose expressions can be evaluated, with respect to a particular memory hierarchy, to compute an estimate of the number of required cache misses. We present a summary of their work, and then extend it to more accurately model total memory traffic in modern systems with larger caches, and when read-only operations can be distinguished from those which modify data.

7.1 Original Model of Manegold, Boncz, and Pirk

Nearly two decades ago, Manegold, Boncz, and Kersten [38] recognized that the majority of database operations could be described in terms of a few simple memory access patterns, such as sequential and random traversals of arrays, random accesses, strided accesses, and so on. The resulting algebra of patterns and their corresponding cost functions formed a *Generic Cost Model for Hierarchical Memory Systems*, designated hereafter as the *Manegold model*. This section reviews their formulation briefly, with slight modifications of notation to accommodate some of the extensions presented below. The extensions of Pirk [45] are also assumed to model traversals with a conditional read probability, as occur in scan-filter operations.

Patterns and Regions

In the Manegold model, every relational table comprises one or more *regions*, each of which has a particular geometry determined by its elements' width and their overall number:

$$\mathtt{R}_1 = \mathsf{Region}\,(b \text{ bytes} \times N \text{ elements})$$

A *basic access pattern* touches all or some elements of one region in some order:

$$\mathtt{P}_1 = \mathsf{S\_Tra}\,(\mathtt{R}_1, u)$$

In the above example, $\mathtt{P}_1$ specifies a sequential traversal of region $\mathtt{R}_1$ that uses u bytes of each element of size $b \geq u$. Table 7.1 summarizes the set of basic patterns. Multiple basic patterns can be executed either concurrently or sequentially, as denoted by the operators $\odot$ and $\oplus$, respectively:

$$\mathtt{P} = \mathtt{P}_1 \odot \mathtt{P}_2 \cdots \odot \mathtt{P}_n = \odot\,(\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_n)$$

Pattern	**Symbol**
Sequential Traversal	$\mathsf{S\_Tra}\,(\mathtt{R}, [u])$
Touches each element of $\mathtt{R}$ in sequence	
Repeated Sequential Traversal	$\mathsf{RS\_Tra}\,(\mathtt{R}, N, \mathbf{dir}, [u])$
N repeated sequential traversals in same ($\mathbf{dir} = \mathbf{uni}$) or alternating **bi** directions	
Random Traversal	$\mathsf{R\_Tra}\,(\mathtt{R}, [u])$
Touches each element of $\mathtt{R}$ once, but in a random order.	
Repeated Random Traversal	$\mathsf{RR\_Tra}\,(\mathtt{R}, N, [u])$
Randomly traverse the elements of $\mathtt{R}$ repeated N times	
Random Accesses	$\mathsf{R\_Acc}\,(\mathtt{R}, N, [u])$
Makes N random accesses to the elements of $\mathtt{R}$	
Conditional Read Sequential	$\mathsf{S\_Tra\_CR}\,(\mathtt{R}, \sigma, [u])$
Sequentially traverses the elements of $\mathtt{R}$, but only accesses each element with probability $\sigma \leq 1$.	
Nested Cursor Pattern	$\mathsf{Nest}\,(\mathtt{P} \in \mathcal{B}, N, \mathbf{rs})$
Divides pattern $\mathtt{P}$ into N simultaneously-active sub-patterns on $1/N$th of $\mathtt{P}$'s region $\mathtt{R}$. If $\mathbf{rs} = \mathbf{rand}$ then sub-patterns are interleaved randomly; they are alternated in sequence if $\mathbf{rs} = \mathbf{seq}$.	
Blocked Sequence Pattern	$\bigoplus\limits_{i \leq N} [\mathtt{P}]$
Similar to $\mathsf{Nest}\,()$, but with sub-patterns repeated N times sequentially.	

Figure 7.1: Basic Patterns in the Manegold Model

Caches and Cache States

The cost of a pattern may depend on the cache hierarchy with which it is executed. A cache of size $\mathcal{C} = \mathit{linesize} \times \#\ \mathit{lines}$ is assumed to operate with full associativity, while an initial cache state $\mathtt{I}_{\mathcal{C}} = \{< \mathtt{R}_i \; : \; l >\}$ specifies the number of lines of each region that reside in cache at the moment when execution begins.

A vector-valued *miss function* gives the number of cache misses induced by pattern $\mathtt{P}$:

$$\mathbf{M}_{\mathcal{C}}(\mathtt{I}, \mathtt{P}) = \langle \textit{sequential}, \; \textit{random} \rangle$$

The miss vector separates *random* misses from *sequential* ones, as a realistic memory technology–or system–is likely to exhibit asymmetric performance in the two cases. At the same time, a state update function $\mathbf{S}(\mathtt{P}, \mathtt{I}_{\mathcal{C}})$ yields the resulting cache state.

As an example, a basic sequential traversal with an empty cache will result in:

$$\mathbf{M}_{\mathcal{C}}\left(\mathsf{S\_Tra}\left(\mathtt{R}\right)\right) = \langle |\mathtt{R}|_{\mathcal{C}}, 0\rangle \; ; \; \mathbf{S}_{\mathcal{C}}\left(\mathsf{S\_Tra}\left(\mathtt{R}\right)\right) = \{\langle \mathtt{R} : \, |\mathcal{C}|\rangle\}$$

Here, $|\mathcal{C}|$ denotes the number of lines in cache $\mathcal{C}$, while $|\mathtt{R}|_{\mathcal{C}}$ is the size of region $\mathtt{R}$ as a number of cache lines of $\mathcal{C}$. For detailed miss functions $\mathbf{M}_{\mathcal{C}}(\mathtt{P} \in \mathcal{B})$ for other basic patterns, the reader is referred to the original paper [38] of Manegold et al. In general, the state update of a basic pattern $\mathtt{P}$ operating on region $\mathtt{R}$ is:

$$\mathbf{S}_{\mathcal{C}}\left(\mathtt{P}\right) = \{\langle \mathtt{R} : \; \min\left(\mathbf{F}_{\mathcal{C}}(\mathtt{P}), |\mathcal{C}|\right)\rangle\} \tag{7.1}$$

The expression $\mathbf{F}_{\mathcal{C}}(\mathtt{P})$ denotes the *footprint* of pattern $\mathtt{P}$, or the total number of cache lines (w.r.t. $\mathcal{C}$) touched by it, which in the case of a basic pattern may or may not comprise its entire region.[1]

Costs of Compound Patterns

A sequence of two or more patterns uses the state update function $\mathbf{S}$ to set the new initial state between each pair of sub-patterns:

$$\mathbf{M}_{\mathcal{C}}\left(\mathtt{I}, \oplus\left[\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_n\right]\right) = \mathbf{M}_{\mathcal{C}}\left(\mathtt{I}, \mathtt{P}_1\right) + \mathbf{M}_{\mathcal{C}}\left(\mathbf{S}_{\mathcal{C}}(\mathtt{I}, \mathtt{P}_1), \oplus\left[\mathtt{P}_2, \ldots, \mathtt{P}_n\right]\right) \tag{7.2}$$

Its corresponding state update is given as

$$\mathbf{S}_{\mathcal{C}}\left(\mathtt{I}, \oplus\left[\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_n\right]\right) = \mathbf{S}_{\mathcal{C}}\left(\mathbf{S}_{\mathcal{C}}\left(\mathtt{I}, \oplus\left[\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_{n-1}\right]\right), \mathtt{P}_n\right) \tag{7.3}$$

In the case of a concurrent pattern, such as $\mathtt{P} = \mathsf{S\_Tra}\left(\mathtt{R}_1\right) \odot \mathsf{R\_Tra}\left(\mathtt{R}_2\right)$, the Manegold model calculates the miss count for each sub-pattern as if it were executed by itself in a cache with a fraction of the size of $\mathcal{C}$. Each pattern is allotted a fraction of available cache space based on its contribution to the overall *working set*, which is the total number of cache lines in active use at any point during execution[2].

[1] This is actually a departure from the Manegold formulation, which merely considers the entire region to be mapped. For $\mathsf{S\_Tra}\left(\mathtt{R}, u \leq \mathtt{R.width}\right)$, we use their formula for miss count to estimate the footprint, and use the equivalent formula from Pirk [45] for $\mathsf{S\_Tra\_CR}\left(\mathtt{R}, \sigma\right)$.

[2] In their terminology, this is a *footprint*, but we reserve that term for the total number of lines touched over the entire course of execution.

In a sequential traversal, only one line at a time is needed, while a random pattern might reuse every line in its *footprint*, or total set of lines touched during execution. The above pattern's cost will thus be computed as:

$$\mathbf{W}_{\mathcal{C}}\left(\mathtt{P}\right) = \mathbf{W}_{\mathcal{C}}\left(\mathsf{S\_Tra}\left(\mathtt{R}_1\right)\right) + \mathbf{W}_{\mathcal{C}}\left(\mathsf{R\_Tra}\left(\mathtt{R}_2\right)\right) = 1 + |\mathtt{R}_2|_{\mathcal{C}}$$

In general, a concurrent pattern's cost is given by

$$\mathbf{M}_{\mathcal{C}}\left(\mathtt{I}, \odot\left[\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_n\right]\right) = \sum_{i \le n} \mathbf{M}_{\mathcal{C}/\nu_i}\left(\mathtt{I}, \mathtt{P}_i\right) \tag{7.4}$$

$$\nu_i = \sum_{j \le n} \mathbf{W}_{\mathcal{C}}(\mathtt{P}_j) / \mathbf{W}_{\mathcal{C}}(\mathtt{P}_i) \tag{7.5}$$

$$\mathbf{S}_{\mathcal{C}}\left(\mathtt{I}, \odot\left[\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_n\right]\right) = \bigcup_{i \le n} \mathbf{S}_{\mathcal{C}/\nu_i}\left(\mathtt{I}, \mathtt{P}_i\right) \tag{7.6}$$

Where $\mathcal{C}/\nu_i$ represents a cache with size $|\mathcal{C}|/\nu_i$.

Together, we take the tuple $\mathcal{M} = \langle \mathbf{S}, \mathbf{F}, \mathbf{M} \rangle$ of operators to constitute the *model*, of which each component may be replaced and refined independently. The next section describes our own contributions to the model, which extend it by one component $\mathbf{A}$, the *allocator*, and by expanding the domain of patterns $\mathcal{P}$ expressible in the language.

7.2 Extensions for Communication Bounds

The goal of the foregoing model was to predict the *number of cache misses* on systems with small caches relative to the size of the relations being processed, which was a good metric by which to choose between different physical operator implementations in a query optimizer. By contrast, our goal is to supply a *lower bound on communication* for hypothetical machines of the future. As such, we have extended the Manegold model in several ways: *we (1.) differentiate between read and write traffic, (2.) assume caches may be arbitrarily large and preserve data across* $\oplus$ *sequences, and (3.) also assume that cache space is optimally managed.* The rest of this section describes how our model extensions capture each of these assumptions.

Extensions to Model Large, Optimal Caches

The Manegold model was designed under the assumption of caches with realistic modes of operation and replacement policies, in which space is divided among patterns according to their relative working set sizes, even though this behavior may be far from optimal. For example, the histogram used in a scatter phase of partitioning might well be small relative to the size of the relation being output, but by comparison would benefit enormously from caching. Thus, a true lower bound on communication assumes optimal division of cache resources between patterns according to their *utility*. We defer to later work a discussion of

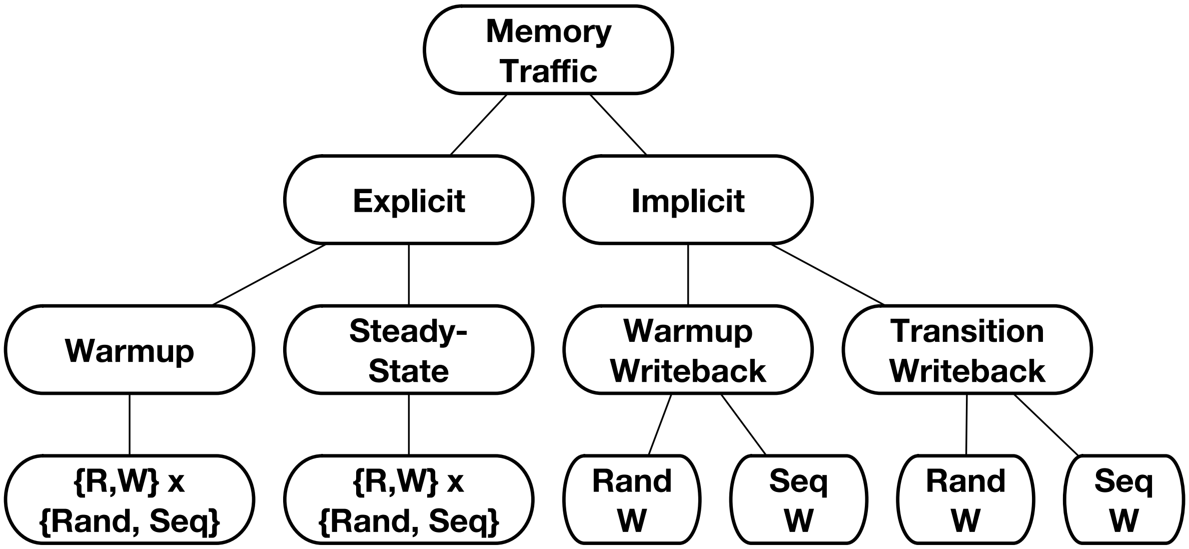


Figure 7.2: Memory Traffic Taxonomy: cache line transfers are either "explicit", meaning they correspond directly to a pattern's per-element accesses, or "implicit", meaning they are implied by the eviction of dirty lines no longer needed.

whether such fine-grain control of memory is feasible in real hardware, but for now augment the Manegold model with a cache *allocation function*, analogous to the cache state update function, and make it a parameter of every cost function.

Cache allocations are similar to initial states, in that they assign to each region a number of cache lines, and are passed as parameters to model operators of the form $\mathbf{M}_{\mathcal{C}}\left(\mathtt{I},\mathtt{A},\mathtt{P}\right)$ or $\mathbf{S}_{\mathcal{C}}\left(\mathtt{I},\mathtt{A},\mathtt{P}\right)$. Thus, the original Manegold model's allocator (a.k.a. the "working set allocator" $\mathbf{A}_{\mathcal{C}}^{\mathbf{W}}$) would simply treat a concurrent pattern $\mathtt{P} = \odot\left[\mathtt{P}_1 \ldots, \mathtt{P}_n\right]$ in the same manner as would the state update function of Equation 7.6.

Optimal Allocation. To illustrate the complexity of the cache allocation problem, we consider only *polynomial* patterns, or those consisting of a sequence $\mathtt{P} = \mathtt{P}_1 \oplus \mathtt{P}_2 \oplus \cdots \oplus \mathtt{P}_n$ of concurrent sub-patterns $\mathtt{P}_i = \mathtt{P}_{i1} \odot \mathtt{P}_{i2} \odot \cdots \odot \mathtt{P}_{im}$ of which each element $\mathtt{P}_{ij} \in \mathcal{B}$ is a basic pattern operating on region $\mathtt{R}_{ij}$. Furthermore, suppose each term of polynomial $\mathtt{P}$ is called a *phase*.

Allocation is thus an integer programming problem of the form:

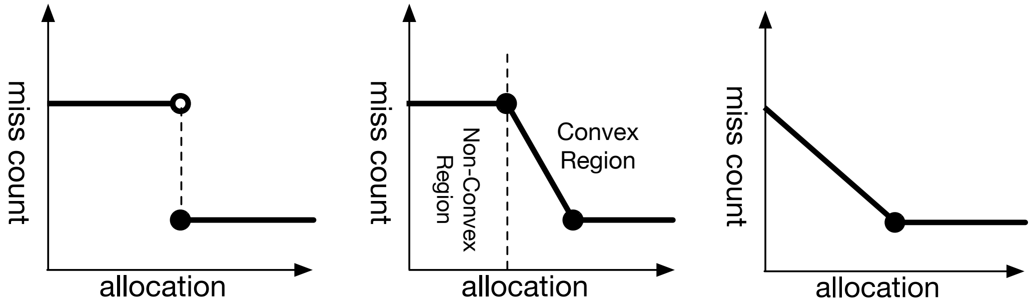
$$\begin{aligned} &\text{Minimize } \mathbf{M}_{\mathcal{C}}\left(\emptyset,\mathtt{A},\mathtt{P}\right) \\ &\text{Subject to } \sum_{\mathtt{P}_{ij}\in\mathtt{P}_i} \mathtt{A}_{ij} \leq |C| \text{ for } \mathtt{P}_i \in \mathtt{P} \end{aligned} \tag{7.7}$$

In this section, we demonstrate that optimal allocation for even a single phase is NP-hard, only in the number of patterns. We present an allocation algorithm that is logarithmic in the size of the cache. We then discuss the more general allocation problem, for which no optimal solution is known, and compare several stopgap solutions.

Single-Phase Allocation We first consider only the *single-phase* allocation problem, in which allocations are produced for the monomial (concurrent) terms of $\mathtt{P}$ independently. Since a single phase consists of a concurrence $\mathtt{P}_i = \odot[\mathtt{P}_{i1}, \ldots, \mathtt{P}_{im}]$ of basic patterns, the objective function of (7.7) reduces to

$$\text{For phase } \mathtt{P}_i\text{: } \underset{\mathtt{A}_{i1},\ldots,\mathtt{A}_{im}}{\operatorname{argmin}} \sum_{j \leq m} \mathbf{M}_{\mathtt{A}_{ij}}(\mathtt{P}_{ij}) \tag{7.8}$$

It is subject to the same constraint. Here, each miss function corresponds to the cost of a basic pattern, each of which has one of three possible shapes. Random patterns have a *convex* curve as shown in Figure 7.3c. Sequential traversals of a region accessed by a previous phase have the *discontinuous* shape, depicted in Figure 7.3a, while repeated sequential traversals (RS_Tra) in the same direction have a continuous, but non-convex form shown in Figure 7.3b. In all cases, the cost function will be a *piecewise linear* function, and so the overall optimization problem corresponds to the *single resource single dimension (SRSD) resource allocation problem* of Lee et al. [32]'s taxonomy. The same authors showed in Rajkumar et al. [54] that this problem is NP-hard in the number of regions; Section 7.6 specifically reduces the single-phase allocation problem to knapsack.



(a) *Discontinuous* (b) *Non-convex* (c) *Convex*

Figure 7.3: Single-Pattern Cost Function Shapes

Nonetheless, we provide a practical solution to these difficulties, which suffices for smaller problems. First, we observe that if all basic pattern cost functions could be made convex, then the polynomial time algorithm of Katoh, Ibaraki, and Mine [24], as cited by Stone et al. [65], could find the optimal allocation for a single phase. Second, we note that the source of non-convexity in each of the Manegold model's patterns is an initial threshold cache amount $\mathsf{conv}_C(\mathtt{P})$, prior to which no benefit is accrued; beyond that, however, the cost function is either flat, or strictly convex. Moreover, this threshold is almost always located at the pattern's footprint $\mathbf{F}(\mathtt{P})$, as in the case of RS_Tra $(N, \mathtt{R}, \mathbf{dir} = \mathbf{uni})$, which has a discontinuity there, benefiting only once the entire region fits in cache, or at one half the footprint size, as in the case of RS_Tra $(N, \mathbf{??dir} = \mathbf{bi})$.

If we were willing to accept an exponential cost in the number of *patterns*–in particular, the number of non-convex patterns–which is likely to be small in any single phase, then we could straightforwardly apply Katoh's *resource allocation procedure (RAP)* algorithm, $\mathbf{A}_{\mathcal{C}}(\mathtt{P}) = \mathtt{RAP}(\mathcal{C}, \mathtt{P})$, in each convex sub-domain. Algorithm 7 does so by considering all possible subsets of non-convex patterns in $\mathtt{P}_i \in \mathtt{P}$, and constraining their allocations to be at least the threshold $\mathsf{conv}(\mathtt{P}_i)$, while rejecting any infeasible allocations $\mathtt{A} > |\mathcal{C}|$. Since there are at most $2^{|\mathtt{P}|}$ such patterns, the runtime of Algorithm 7 is $O(2^{|\mathtt{P}|}(\log|\mathcal{C}|)^2)$, remaining polylogarithmic in the number of allocated cache lines, which is likely to be many orders of magnitude larger than the number of patterns.

Algorithm 7 Pseudo-Convex Cache Allocation

```
function PC-RAP($\mathcal{C}$, $\mathtt{P} = \mathtt{P}_1 \odot \mathtt{P}_2 \cdots \odot \mathtt{P}_n$)
    $BestAlloc \leftarrow \emptyset$
    $MinCost \leftarrow \mathbf{M}_{\mathcal{C}}(BestAlloc, \mathtt{P})$
    $NonConv \leftarrow$ all non-convex basic patterns in $\mathtt{P}$
    $Conv \leftarrow$ all convex basic sub-patterns of $\mathtt{P}$
    for all $P' \in 2^{NonConv}$ do
        $\mathtt{A} \leftarrow \emptyset$                          ▷ Alloc. $P'$ to threshold or more
        for all patterns $\mathtt{P}_i \in P'$ do
            $\mathtt{A}_i \leftarrow \mathsf{conv}_{\mathcal{C}}(\mathtt{P}_i)$           ▷ Set to convex threshold
        if $|\mathtt{A}| > |\mathcal{C}|$ then                    ▷ Don't alloc more than $\mathcal{C}$
            continue                                 ▷ Reject $P'$
        $\mathtt{A} \leftarrow \mathtt{A} \bigcup \mathrm{RAP}(\mathcal{C}, \mathtt{A}, Conv)$
        $Cost \leftarrow \mathbf{M}_{\mathcal{C}}(\mathtt{A}, \mathtt{P})$
        if $Cost < MinCost$ then
            $BestAlloc \leftarrow \mathtt{A}$
            $MinCost \leftarrow Cost$
return BestAlloc
```

For excessively complex patterns with too many regions for brute-force search, the complexity could be reduced to $O(|\mathtt{P}|^2(\log|\mathcal{C}|)^2)$ by replacing each cost function with its *convex hull*, and skipping the $PC - RAP(\dots)$ step, which would induce an error of at most 50% relative to the underlying cost function (as Section 7.7 shows), though this itself is already a first-order approximation of what a real workload and cache would produce. [3]

[3] It may in the end turn out worthwhile to abandon this constraint and use the convex hull for such patterns after all: if a sufficiently "smart" caching policy were used, then some arbitrarily small number of lines could be pinned in cache, and used whenever the part of traversal to which they correspond is reached. This would result in a more continuous marginal utility function for those patterns, and thus eliminate the headaches of non-convexity.

Read-Write Traffic Model

The number of cache misses induced by a particular query plan is a good predictor of its runtime on general-purpose processors, but it is an inadequate estimate of purely bandwidth-bound performance on yet-to-be designed machines. We therefore extend the Manegold model to track both *read and write* operations, and account for the added bandwidth cost of writing back modified (dirty) cache lines. Although this may seem a strenuous effort to correct what is at most an error of 2x, it is crucial for analyzing the benefits of new memory technologies that may have divergent read and write costs. And, factors of two add up quickly.

Discard, Read, Write, and Allocate Flags To separate both kinds of traffic, we extend each basic pattern type from the Manegold model with read-write flags to indicate whether it modifies data it touches. A separate *allocate* flag determines whether a pattern should occupy space in the cache hierarchy, while a *discard* flag indicates whether data modified by this pattern can simply be dropped afterwards, rather than written back to the next level of memory. Thus, a sequential traversal may have the following form:

$$\underset{\mathsf{DRAW}}{\mathsf{S\_Tra}}\,(\mathtt{R})$$

The read-allocate combination corresponds to the implicit assumptions of the Manegold model. A write-only allocate pattern does not result in any misses until the cache is full, while a read-write-allocate pattern exhibits read-only misses initially and then read-write misses at capacity. In a non-allocate pattern, every access will result in a miss.

In general, patterns can have any combination of flags as long as either read or write is set, but the $\mathtt{P}_1 = \mathsf{Nest}\,(\mathtt{P}_2)$ pattern requires special consideration. The flags of the outer pattern $\mathtt{P}_1$ can differ from those of inner pattern $\mathtt{P}_2$ in that the outer nesting must read data whenever the inner pattern writes, even if $\mathtt{P}_2$ is read-only. That particular combination captures a common optimization used in partitioning, as described in Section 7.5.

As an extension of the discard flag, we also introduce a new pattern to mark regions' cached, dirty state as no longer needed:

$$\mathsf{Discard}\,\{\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_n\}$$

Warmup vs. Steady-State Transfers
Handling read/write/allocate flags correctly requires further distinguishing *warmup* accesses–which occur when a pattern has not yet populated its portion of the cache with data–from *steady-state* accesses, which occur when a pattern is fully utilizing its cache allocation. Note that this is *not* identical to the traditional compulsory-capacity distinction: in the latter, the traffic from a sequential read of a yet uncached-region will consist entirely of compulsory misses, while only the first several of those will be warm-ups.

Warmup accesses differ from their steady-state counterparts in that write-allocate patterns do not generate write traffic during the warmup phase. Thus, a pattern whose footprint fits within its cache allocation will will not generate write traffic until its dirty lines are evicted

Pattern	**Warmup**		**Steady**	
	Rnd	**Sq**	**Rnd**	**Sq**
$\text{S\_Tra}_{\text{W}}(...)$		W		
$\text{S\_Tra}_{\text{WA}}(...)$		W	W	W
$\text{S\_Tra}_{\text{RWA}}(...)$		R,W	W	R,W
$\text{S\_Tra}_{\text{RA}}(...)$		R		R
$\text{S\_Tra}_{\text{R}}(...)$		R		
$\text{S\_Tra}_{\text{RW}}(...)$		R,W		

Table 7.1: Read & Write Flags for Sequential Patterns

Pattern	**Warmup**		**Steady**		**Warmup**		**Steady**	
	$\mathtt{R}.u < B$				$\mathtt{R}.u = B$			
	R	**S**	**R**	**S**	**R**	**S**	**R**	**S**
$\text{R\_Tra}_{\text{W}}(...)$	R,W				W			
$\text{R\_Tra}_{\text{WA}}(...)$	R,W		R,W	W		R,W		
$\text{R\_Tra}_{\text{RWA}}(...)$	R,W		R,W	R,W		R,W		
$\text{R\_Tra}_{\text{RA}}(...)$	R		R	R		R		
$\text{R\_Tra}_{\text{R}}(...)$	R			R				
$\text{R\_Tra}_{\text{RW}}(...)$	R,W			R,W				

Table 7.2: Read & Write Flags for Random Patterns

in a phase transition. Tables 7.1 and 7.2 indicate which patterns generate read and write traffic in the warmup and steady-state phases for sequential and random traffic, respectively.

A pattern $\mathtt{P}$ that causes state transition $\mathtt{I} \rightarrow \mathtt{S2} = \mathbf{S}_{\mathcal{C}}(\mathtt{I}, \mathtt{A}, \mathtt{P})$ will therefore have at least the number of warmup misses in Equation 7.9, where $\mathsf{IUtil}\,(\ldots)$ determines whether $\mathtt{P}$ can benefit from initial state, and $\mathsf{acc}(\mathtt{P})$ is the number of accesses it generates:

$$\mathsf{warm}_{\mathcal{C}}(\mathtt{I}, \mathtt{A}, \mathtt{P}) = \begin{cases} \mathsf{acc}(\mathtt{P}) & \neg\mathsf{alloc}\,(\mathtt{P}) \\ \max\left(0, \Delta\left(\mathtt{I}, \mathtt{A}, \mathtt{P}\right)\right) & \mathsf{alloc}\,(\mathtt{P}) \end{cases} \tag{7.9}$$

$$\Delta\left(\mathtt{I}, \mathtt{A}, \mathtt{P}\right) = \min\left(\mathtt{A[R]}, \mathbf{F}_{\mathcal{C}}(\mathtt{P})\right) - \left(\mathsf{IUtil}\left(\mathtt{I}, \mathtt{A}, \mathtt{P}\right)\right)\left(\mathtt{I[R].l}\right) \tag{7.10}$$

All misses (random and sequential) beyond that number are classified as steady-state, and are designated as read, write, or read-write according to the "steady" column of Tables 7.1 and 7.2. In the special case where a pattern does not set the allocate flag, all misses are considered warm-ups. Since some patterns, such as sequential traversals of regions larger than the cache, are assumed not to benefit from initial state, we set the initial utility function as

$$\mathsf{IUtil}\left(\mathtt{I}, \mathtt{A}, \mathtt{P} \in \mathcal{B}\right) = \begin{cases} 1 & \mathsf{rand}(\mathtt{P}) \\ 1 & \mathtt{I[R].l} = |\mathtt{R}|_{\mathcal{C}} \\ 0 & \text{otherwise} \end{cases} \tag{7.11}$$

Steady-State Write-backs. The number of lines $\mathbf{M}_{\mathcal{C}}^{SSW}(\mathtt{I},\mathtt{A},\mathtt{P})$ transferred because of steady-state write-backs is equal to the number of misses caused by patterns that write data, minus the number of warmup misses given by Equation 7.9 above. The number of steady-state write-backs for basic pattern P is thus given as:

$$\mathbf{M}_{\mathcal{C}}^{SSW}(\mathtt{I},\mathtt{A},\mathtt{P}) = \max\left\{0, \mathbf{M}_{\mathcal{C}}(\mathtt{I},\mathtt{A},\mathtt{P}) \cdot \mathsf{write}\left(\mathtt{P}\right) - \mathsf{warm}_{\mathcal{C}}(\mathtt{I},\mathtt{A},\mathtt{R})\right\} \tag{7.12}$$

For a sequence of two or more patterns, the number of explicit write-backs is defined as:

$$\begin{aligned} &\mathbf{M}_{\mathcal{C}}^{SSW}\left(\mathtt{I},\mathtt{A},\oplus[\mathtt{P}_1,\ldots,\mathtt{P}_n]\right) = \\ &\quad \mathbf{M}_{\mathcal{C}}^{SSW}(\mathtt{I},\mathtt{A}_1,\mathtt{P}_1) + \mathbf{M}_{\mathcal{C}}^{SSW}\left(\mathbf{S}(\mathtt{I},\mathtt{A},\mathtt{P}_1),\mathtt{A},\oplus[\mathtt{P}_2,\ldots,\mathtt{P}_n]\right) \end{aligned} \tag{7.13}$$

The formula for concurrent write misses is analogous.

Implicit Traffic: Line Eviction Write-backs

Implicit traffic occurs when cached data has been modified with respect to the next level of storage, and is evicted from cache because it is no longer being used. Implicit write-backs differ from others in that they do not occur during the steady state. We classify such transfers as *transition* misses, implied when fewer lines of a region are cached after a state transition, and *warmup* misses, which occur when a single pattern cannot be assumed to make use of lines from its own region that are already cached. We also assume that data read into the cache sequentially can be written out sequentially. To track both of these distinctions, we extend the cache state model with two new per-region attributes, and present formulas below which use them.

Tracking Dirty State Counting transition misses requires an expansion of the cache state model to indicate not only how much of each region is present, but also which fraction of that is *dirty*, or modified with respect to the next level of memory. For a basic pattern with region R, we modify the state update formula as follows, where I[R].l denotes the number of lines of R in cache, while the subscript d denotes the number of lines that are dirty:

$$\begin{aligned} \mathbf{S}_{\mathcal{C}}(\mathtt{I},\mathtt{P}(R)\in\mathcal{B}) &= \{\langle \mathtt{R}:\mathtt{l}=\min(\mathbf{F}_{\mathcal{C}}(\mathtt{P}),|\mathcal{C}|),\mathtt{d}=D(\mathtt{I},\mathtt{P})\rangle\} \\ D(\mathtt{I},\mathtt{P}) &= \begin{cases} \min\left(\mathbf{F}(\mathtt{P}),|\mathcal{C}|\right) & \mathtt{P.write} \\ \min\left(\mathtt{I[R].l},|\mathcal{C}|\right) & \text{otherwise} \end{cases} \end{aligned} \tag{7.14}$$

In a sequential pattern $\mathtt{P}_1 \oplus \mathtt{P}_2$, the transition between states $\mathtt{S1} \rightarrow \mathtt{S2}$ generates traffic equal to the net decrease in the number of cached, dirty lines:

$$\mathbf{M}_{\mathcal{C}}^{TWB}(\mathtt{I},\mathtt{A},\mathtt{P}_1\oplus\mathtt{P}_2) = \sum_{\mathtt{R}\in\mathtt{S1}} \max\left(0,[\mathtt{S1[R].d}] - [\mathtt{S2[R].d}]\right)$$

Warmup Write-backs In some cases, the initial cache state may contain some lines of a basic pattern's region, but it will not be possible to make use of those lines before they are evicted, as determined by $\mathsf{lUtil}\,()$. If some of those lines are dirty, they will need to be written back, as accounted for the by the *warmup write-back* function $\mathsf{WWB}_{\mathtt{R}}$ for a single region:

$$\mathsf{WWB}_{\mathtt{R}}(\mathtt{I}, \mathtt{P} \in \mathcal{B}) = (1 - \mathsf{lUtil}\,(\mathtt{P}, \mathtt{I}))(\mathtt{I}\,[\mathtt{R}]\,.\mathtt{d}) \tag{7.15}$$

In a concurrent pattern, the total warmup write-back traffic is given by summing over the individual write-back traffic for each region:

$$\mathbf{M}_{\mathcal{C}}^{WWB}\,(\mathtt{I}, \mathtt{A}, \odot\,[\mathtt{P}_1, \mathtt{P}_2, \ldots, \mathtt{P}_n,]) = \sum_{\mathtt{R}_i \in \mathtt{P}} \left[\max_{\mathtt{P}_j \in \mathtt{P}} \mathsf{WWB}_{\mathtt{R}_i}(\mathtt{I}, \mathtt{P}_j)\right] \tag{7.16}$$

Random vs. Sequential Write-backs Just as the read- and write-bandwidths of a given memory technology may differ, so too may its performance for random and sequential workloads. In order to maintain this distinction for write-back transfers, it is necessary to mark each region in a cache state $\mathbf{S}$ with a *random fraction* $\mathbf{S}(\mathtt{R}).\mathtt{r}$ indicating which portion of its dirty lines are to be counted as random write-backs, with all others resulting in sequential writes. Thus, Equation (7.14) is augmented to stipulate that $\mathbf{S}(\mathtt{P}) = \{\langle \mathtt{R}, \ldots, \mathtt{r} = \mathsf{rand}(\mathtt{P})\rangle\}$.

Block-wise Patterns

Partitioning is frequently employed to speed up sorting, joins, and other operations by constraining the range over which random accesses are distributed, resulting in fewer cache misses. The $\mathsf{Nest}\,(\ldots)$ pattern originally introduced in [38] captures the access pattern of partitioning itself, but more effort is required to model subsequent operations that process one partition or *block* at a time.

Manegold et al. represent block-wise operation notationally as $\oplus|_{j=1}^{m}(\mathsf{hash\_join}(V_j, U_j, W_j))$, for example, in the case of a radix-m join. They do not specify how to compute its cost, because in their framework it would only require multiplying the cost of a single block by the number of blocks m (and replacing each subscripted region by one shrunk by the same factor). However, our proposed extensions necessitate greater subtlety to account for inter-block and inter-phase write-backs.

We introduce new notation for block-wise execution:

$$\underset{m,\mathbf{R}}{\mathsf{Block}}\,(\mathtt{P} \in \mathcal{P})$$

The set $\mathbf{R} = \{\mathtt{R}_1, \mathtt{R}_2, \ldots\}$ of regions is said to be *blocked* meaning that any basic patterns accessing them will be assumed to operate on m sub-regions of size $|\mathtt{R}_i|/m$; any pattern whose region is not in $\mathbf{R}$ is assumed to be *repeated m times*. Additionally, the notation $\mathtt{P}/m$ stands for replacement of any regions in $\mathtt{P}$ by equivalents with $1/m$th the number of elements.

Blocked Basic Patterns The explicit communication cost of a basic pattern depends on whether its region is in the blocked set $\mathbf{R}$ and whether it fits in cache. Indeed, unless the whole unblocked region is already in cache, the blocked pattern is assumed not to benefit from any initially cached data. These distinctions are encoded in the *single block cost* and *blocked initial state* functions, respectively:

$$
{}^{m}_{\mathbf{R}}\left[\mathbf{M}_{\mathcal{C}}\right](\mathtt{I},\mathtt{A},\mathtt{P}(\mathtt{R})\in\mathcal{B}) = \begin{cases} \mathbf{M}_{\mathtt{A}_{\mathtt{R}}}\left(\mathsf{IBlk}_{\mathtt{A}_{\mathtt{R}}}(\mathtt{I},\mathtt{P}),\mathtt{P}/m\right) & \mathtt{R}\in\mathbf{R} \\ \mathbf{M}_{\mathtt{A}_{\mathtt{R}}}(\mathtt{I},\mathtt{P}) & \mathtt{R}\notin\mathbf{R} \end{cases} \tag{7.17}
$$

$$
\mathsf{IBlk}_{\mathcal{C}}(\mathtt{I},\mathtt{P}) = \begin{cases} \emptyset & \mathtt{I[R].l} < |\mathtt{R}|_{\mathcal{C}} \\ \langle\{\mathtt{R}:\mathtt{I[R]}\}\rangle & \mathtt{I[R].l} = |\mathtt{R}|_{\mathcal{C}} \end{cases} \tag{7.18}
$$

The total cost of a basic pattern with m-way blocking is simply m times the single block cost, as non-blocked patterns will be repeated m times too:

$$
\mathbf{M}_{\mathcal{C}}(\mathtt{I},\mathtt{A},\underset{m,\mathbf{R}}{\mathsf{Block}}(\mathtt{P}\in\mathcal{B})) = m\cdot\left({}^{m}_{\mathbf{R}}\left[\mathbf{M}_{\mathcal{C}}\right](\mathtt{I},\mathtt{A},\mathtt{P})\right) \tag{7.19}
$$

Blocked Compound Patterns A concurrent pattern's single-block cost is simply the sum of the costs of its sub-patterns' *blocked variants*:

$$
{}^{m}_{\mathbf{R}}\left[\mathbf{M}_{\mathcal{C}}\right](\mathtt{I},\mathtt{A},\mathtt{P}_1\odot\ldots\mathtt{P}_n) = \sum_{i\leq n}{}^{m}_{\mathbf{R}}\left[\mathbf{M}_{\mathcal{C}}\right](\mathtt{I},\mathtt{A}_i,\mathtt{P}_i) \tag{7.20}
$$

The cost of blocked sequences is computed similarly to normal sequence costs in (7.2), but with recursive blocking of sub-patterns and initial states, and with a special provision for supplying the resulting cache state of $\mathtt{P}_n$ as the initial state for $m-1$ repetitions of $\mathtt{P}_1$:

$$
\begin{aligned}
&\mathbf{M}^{X}_{\mathcal{C}}(\mathtt{I},\mathtt{A},\mathtt{P}_1\oplus\cdots\oplus\mathtt{P}_n) = \\
&\quad\left(\sum_{i=2}^{n}{}^{m}_{\mathbf{R}}\left[\mathbf{M}^{X}_{\mathcal{C}}\right](S_{i-1},\mathtt{A}_i,\mathtt{P}_i)\right)\cdot m + \left({}^{m}_{\mathbf{R}}\left[\mathbf{M}^{X}_{\mathcal{C}}\right](S_n,\mathtt{A}_1,\mathtt{P}_1)\right)\cdot(m-1) + {}^{m}_{\mathbf{R}}\left[\mathbf{M}^{X}_{\mathcal{C}}\right](\mathtt{I},\mathtt{A}_1\mathtt{P}_1)
\end{aligned} \tag{7.21}
$$

Here, the state sequence $S_0\ldots S_n$ is defined through the recurrence

$$
S_0 = \mathtt{I};\qquad S_i = \mathbf{S}\left(S_{i-1},\mathtt{A}_i,\mathtt{P}_i/m\right) \tag{7.22}
$$

Write Traffic in Blocked Patterns In (7.21), the miss function is superscripted with an X to indicate that the same template also defines blocked versions of the warmup, transition, and steady-state write-back cost models. Within the leaf expressions ${}^{m}_{\mathbf{R}}\left[\mathbf{M}_{\mathcal{C}}\right]$, any function $f_{\mathcal{C}}(\mathtt{I},\mathtt{A},\mathtt{P})$ applied to basic patterns, such as $\mathsf{warm}_{\mathcal{C}}(\ldots)$ (7.9), or $\mathsf{WWB}_{\mathtt{R}}(\mathtt{I},\mathtt{P})$ (7.15), are blocked analogously to $\mathsf{IBlk}_{\mathcal{C}}(\ldots)$:

$$
{}^{m}_{\mathbf{R}}\left[f_{\mathcal{C}}\right](\mathtt{I},\mathtt{A},\mathtt{P}(\mathtt{R})) = \begin{cases} f_{\mathcal{C}}\left(\mathsf{IBlk}_{\mathcal{C}}(\mathtt{I},\mathtt{P}),\mathtt{A},\mathtt{R}/m\right) & \mathtt{R}\in\mathbf{R} \\ f_{\mathcal{C}}(\mathtt{I},\mathtt{A},\mathtt{P}) & \mathtt{R}\notin\mathbf{R} \end{cases} \tag{7.23}
$$

7.3 A Manegold-Based Roofline Model

In addition to modeling raw traffic, the point of the foregoing should be to model limits on *performance*. Chapter 1 highlights the Roofline Model as a characterization of machine performance limits in terms of memory bandwidth and arithmetic bandwidth; it is therefore natural to consider limits on the latter in tandem with any estimation of the former. This gives rise to a wide spectrum of modeling granularities and accuracies that may be useful in different contexts. We define four different levels of performance modeling, increasing in fidelity from pure communication costs to full microarchitectural simulation, and consider what each one can offer designers of hypothetical query-processing hardware.

1. **Bandwidth-only (BWOnly)**: Performance is strictly determined by the amount of data transferred between levels of the memory hierarchy. Computation is assumed to be infinitely fast, and memory requests are generated at the maximum rate supported by the memory controller.

2. **Constant access rate (ConstRate)**: Computational power is no longer infinite, and an abstract machine is assumed to be capable of generating a fixed number of memory accesses per second. If the number of misses generated per access is low, the pattern is considered to be compute-bound.

3. **Approximate Arithmetic Intensity (ArInt)**: Each basic operation (e.g. computing a record's hash, incrementing a counter) is assigned an abstract *number of arithmetic/logic operations (AOps)*, and each phase of a query plan requires a certain number AOps per access (instead of one per access, as in ConstRate).

   The relevant notion of "operation" will depend on the kind of arithmetic units envisioned by the designer.

4. **Cycle-accurate Simulation (CycAcc)**: Performance is defined by the actual execution of real code on a concrete hardware design. In a general-purpose processor design, for example, overheads from inter-instruction dependencies, sub-optimal cache replacement policies, inaccurate prefetching, and programmer error (e.g. failure to use streaming stores), will all impact performance.

In Simulation Level 1 (BWOnly), the model described in the preceding section is sufficient to state an absolute lower bound on runtime for a given plan and platform:

$$\overset{BWOnly}{\text{time}}_{\mathcal{C}}(\mathtt{P}) = \frac{\text{MemBW}_{\mathcal{C}}}{\mathbf{M}_{\mathcal{C}}(\mathtt{P}) \cdot B} \tag{7.24}$$

Here, $\text{MemBW}_{\mathcal{C}}$ stands for the peak sustainable bandwidth between cache $\mathcal{C}$ and the next layer of the memory hierarchy. In practice, the read, write, and read-write bandwidths can differ widely, so it is necessary to separate the miss vector and machine bandwidth tuple into their various components. Overall runtime is the maximum of any component of (7.24).

At all higher-fidelity levels, it is also necessary to account for the rate at which memory requests are generated, as not all available bandwidth may be used. In Level 2 (ConstRate), a maximum *access bandwidth* AccBW of accesses per second imposes an additional limit on performance. If $\mathtt{P}$ is a basic pattern, and $\mathsf{acc}(\mathtt{P})$ is the number of memory accesses it generates, then:

$$\overset{\textit{ConstRate}}{\text{time}}_{\mathcal{C}}\,(\mathtt{P} \in \mathcal{B}) = \max\left\{\frac{\text{MemBW}_{\mathcal{C}}}{\mathbf{M}_{\mathcal{C}}(\mathtt{P}) \cdot B}, \frac{\text{AccBW}}{\mathsf{acc}(\mathtt{P})}\right\} \tag{7.25}$$

The access bandwidth is independent of cache level, as it is assumed to depend only on the kind and number of processing elements.

When $\mathtt{P}$ is a compound pattern, however, the calculus is more complex, because a sequence of multiple patterns may contain some that are bandwidth-bound and some that are compute-bound; since *max* is an inherently non-linear operator, it would be inappropriate to apply Equation 7.25 across the average of these patterns. We therefore divide each complex pattern into distinct *phases*, where the top-level sequence $\mathtt{P} = \oplus\,[\mathtt{P}_1, \ldots, \mathtt{P}_n]$ defines phases $1 \ldots n$:

$$\overset{\textit{ConstRate}}{\text{time}}_{\mathcal{C}}\,(\oplus\,[\mathtt{P}_1, \ldots, \mathtt{P}_n]) = \sum_{i \leq n} \overset{\textit{ConstRate}}{\text{time}}_{\mathcal{C}}(\mathtt{P}_i)$$

Note that we perform this separation *only once*, assuming that the pattern is a polynomial, and that its top level is a sequence $\mathtt{P}_1 \oplus \mathtt{P}_2 \ldots \mathtt{P}_n$.[4] Furthermore, the number of accesses of any complex pattern $\mathsf{acc}(\mathtt{P})$ is simply the sum of the accesses of all its constituent basic patterns.

Finally, the third simulation level ArInt trades the access bandwidth term of Equation 7.25 for one constrained by arithmetic operation bandwidth:

$$\overset{\textit{ArInt}}{\text{time}}_{\mathcal{C}}(\mathtt{P}) = \max\left\{\frac{\text{MemBW}}{\mathbf{M}_{\mathcal{C}}(\mathtt{P}) \cdot B}, \frac{\text{AOpBW}}{\text{AOps}(\mathtt{P})}\right\} \tag{7.26}$$

7.4 Summary & Limitations

These extensions to the Manegold model attempt to adapt it for the context of contemporary and emerging memory system architectures, which may include features such as HBM that effectively act as multi-gigabyte caches, or which allow for greater control over allocation, and data write-back. Through their use in the evaluation of broad spaces of query plans, designers of future database machines should be able to analyze the marginal impacts of additional resources such as on-chip storage on query performance limits, without necessarily executing concrete software on a cycle-level simulation.

[4] We rely on the programmer/query compiler to make this segmentation appropriately. While it would be possible to further sub-divide each concurrent pattern into phases based on any sequences it may itself contain (and based on some notion of relative rates of completion of concurrent patterns), this is beyond the scope of the Manegold model, and of unjustifiable fidelity, given that the latter assumes cache allocations based on the maximum working set size of any element of a sequence contained within a concurrence.

That said, it is hard to judge the adequacy of these proposals in the absence of empirical validation or even detailed simulation results, and it is easy to identify some sources of weakness. Even under the common but unrealistic assumption of uniformly distributed accesses in any random ($\mathsf{R\_}$) pattern, the miss functions used to model them are gross simplifications of a complex interaction (even more so in our own implementation, which makes the combinatorially incorrect equation of hit rate with the ratio of a region's size to its allocation). Furthermore, complex queries are likely to present concurrent patterns that access the same region (especially if those patterns are generated automatically), and the model cannot distinguish between constructive and destructive interference (i.e. whether such patterns compete for bandwidth and cache space or share data). The cache allocation algorithm described above is only optimal within a particular stage of a polynomial pattern, and only relative to the underlying miss functions–all of which are limitations that may badly limit utility for real-world queries and datasets.

Nonetheless, these preliminary considerations constitute a step towards the establishment of a more widespread convention of bandwidth-bound performance analysis in the realm of analytics acceleration. We conclude with a brief presentation of how a space of fully-compiled plans might be encoded in such a model. At present, this is done manually, but one motivation for the development of Ressort was to automate this analysis in the future.

7.5 Case Study: TPC-H Query 19

To illustrate how this system models communication requirements for real queries, TPC-H Q19 once again serves as an example. As the machine-level plan study in Chapter 2 Section 2.4 shows, this query consists of (1.) a selection of the `lineitem` table based on three attributes, (2.) a selection of records in the `part` table based on two attributes, (3.) a foreign-key join of the `lineitem` table with the `part` table on the key `partkey`, and (4.) a final filter based on several other attributes of both relations, followed by an aggregation to a single scalar value. With this as the basic plan configuration, other machine-level plans–each with different traffic requirements–are derived according to the processes described in Section 2.2, and a handful of them are sampled from the space also described in that case study. As in prior chapters, it is also assumed that all attributes are stored in column-major order, and that discrete quantities such as `p_container` are stored as dictionary-encoded single-byte values. However, in this case, we do not consider parallelization, and focus only on serial plans. For brevity, `lineitem` is abbreviated as L and `part` as P.

The `lineitem` table is first filtered based on on the predicates involving the $L^{\text{pre}} = \{\texttt{l\_shipmode}, \texttt{l\_shipinstruct}\}$ attributes (selectivity $\sigma_L = 0.083$ for this query) the results of which are subsequently joined against the `part` table, followed by a final selection based on the remaining attributes of both tables:

$$L^{\text{post}} = \{\texttt{l\_quantity}, \texttt{l\_extendedprice}, \texttt{l\_discount}\}$$
$$P^{\text{post}} = \{\texttt{p\_size}, \texttt{p\_container}, \texttt{p\_brand}\}$$

At the highest level, the space of machine-level plans is divided along two dimensions: (1.) partitioning (based on `partkey`) of zero, one, or all attributes, (2.) early vs. late materialization of column-major data into multi-attribute output records.

Thus, each plan's access pattern can be described as a combination of two halves (each of which may constitute multiple "phases" in the sense of loops, or terms of a polynomial pattern): a *partition* phase, and a *join* phase. The patterns of each phase can be described separately.

Partitioning

Assume that partitioning entails division into 2^m buckets based on the m most significant bits (MSBs) of the value of the `partkey` attribute. In all cases, this entails a *histogram* data structure of size:

$$\mathrm{Hist}_X = \mathsf{Region}\left(2^m \times \frac{1}{8}\log_2|X|\right) \tag{7.27}$$

Building that histogram requires the same access patterns for all plans:

$$\begin{aligned}
\mathrm{PBuild}(L) = {} & \left[\bigodot_{\mathtt{r}\in L^{\mathrm{pre}}} \underset{\mathsf{R}}{\mathsf{S\_Tra}}\left(\mathtt{r}\right)\right] \odot \underset{\mathsf{WA}}{\mathsf{S\_Tra}}\left(\mathrm{BVec}\right) \\
& \odot \underset{\mathsf{R}}{\mathsf{S\_Tra\_CR}}\left(L_{\mathrm{key}}, \sigma_L\right) \\
& \odot \underset{\mathsf{RW}}{\mathsf{R\_Acc}}\left(\mathrm{Hist}_L, |L|\sigma_L\right) \\
\mathrm{PBuild}(P) = {} & \underset{\mathsf{R}}{\mathsf{S\_Tra}}\left(P_{\mathrm{key}}\right) \odot \underset{\mathsf{RW}}{\mathsf{R\_Acc}}\left(\mathrm{Hist}_P, |P|\right)
\end{aligned} \tag{7.28}$$

In the move phase, common optimization of write-combining buffers [59, 5, 37] is assumed in the model, meaning an in-cache buffer is presumed to be allocated with one cache line per output partition (in case the output record size is smaller than a cache line, which it is in this case, and that the buffer would fit in cache), allowing that line to fill completely before being evicted to to its proper place in the memory-resident output array:

$$\mathrm{PWbMv}(X) = \underset{\mathsf{RWA}}{\mathsf{Nest}}\left(2^m, \underset{\mathsf{WA}}{\mathsf{S\_Tra}}\left(\widehat{X}\right), \mathsf{rand}\right) \tag{7.29}$$

Here, the outer nesting has both read *and* write flags set, while the inner sequential traversal has only the write flag; as a consequence, read traffic is generated only for misses to the 2^m cache lines that constitute the working set of $\underset{\mathsf{RWA}}{\mathsf{Nest}}(\dots)$, while $\underset{\mathsf{WA}}{\mathsf{S\_Tra}}(X)$ generates writes only.

In a partition-all plan, the model assumes that it is possible to avoid re-scanning the pre-join attributes L^{pre} each time a new attribute of L is partitioned, because a *bit vector* BVec can be employed to indicate whether each element of L was selected during the initial $\mathrm{PBuild}(L)$ scan (though other plans are possible). For each subsequent attribute, the keys of L are re-scanned as (for P, $\text{PScan}(P) = \text{PBuild}(P)$):

$$\begin{aligned} \text{BVec} &= \textsf{Region}\,(|L| \times 1/8) \\ \text{PScan}(L) &= \underset{\textsf{R}}{\textsf{S\_Tra}}\,(\text{BVec}) \odot \underset{\textsf{R}}{\textsf{S\_Tra\_CR}}\,(L_{\text{key}}, \sigma_L) \\ &\qquad\qquad \odot \underset{\textsf{RWA}}{\textsf{R\_Acc}}\,(\text{Hist}_L, |L|\sigma_L) \end{aligned} \tag{7.30}$$

The partitioning phase thus encompasses histogram *build* and record *move* sub-phases for both relations:

$$\text{Part} = \text{PBuild}(P) + \text{PMove}(P) + \text{PBuild}(L) + \text{PMove}(L) \tag{7.31}$$

Part-single-attr Partitioning on a single attribute entails a simple four-phase build-move-build-move sequence. We denote by $\widehat{X}$ the partitioned version of region X.

$$\begin{aligned} \widehat{X}_{\text{key}} &= \textsf{Region}\,(|X|\sigma_X \times [\mathit{KeySize} \cdot \mathit{PtrSize}]) \\ \text{PMove}(X) &= \text{PScan}(X) \odot \text{PWbMv}(X_{\text{key}}) \end{aligned}$$

Part-all-late Partitioning all attributes introduces the option of materializing records early, either during the partitioning itself, or while the hash table is built during the join. We consider now the first two cases, leaving the latter until we consider the join phase itself. When no materialization is performed, the partition-all pattern has the same build-phase behavior as no-partition, but the move phase is given as:

$$\begin{aligned} \widehat{L}_{\text{key}} &= \textsf{Region}\,(|L|\sigma_L \times [L_{\text{key}}\texttt{.width} + \mathit{PtrSize}]) \\ \widehat{L}_{i\neq\text{key}} &= \textsf{Region}\,(|L|\sigma_L \times L_i\texttt{.width}) \\ \text{PMove}^{\text{val}}(P, \texttt{r}) &= \text{PScan}(P) \odot \text{PWbMv}(\texttt{r}) \odot \underset{\textsf{R}}{\textsf{S\_Tra}}\,(\texttt{r}) \\ \text{PMove}^{\text{val}}(L, \texttt{r}) &= \text{PScan}(L) \odot \text{PWbMv}(\texttt{r}) \odot \underset{\textsf{R}}{\textsf{S\_Tra\_CR}}\,(\texttt{r}, \sigma_L) \\ \text{PMove}^{\text{key}}(X) &= \text{PScan}(X) \odot \text{PWbMv}(X_{\text{key}}) \\ \text{PMove}(X) &= \bigoplus_{\texttt{r}\in X^{\text{post}}} \left[\text{PMove}^{\text{val}}(X, \texttt{r})\right] \oplus \text{PMove}^{\text{key}}(X) \end{aligned}$$

Part-all-early Materializing attribute values from each relation during the move phase of partitioning results in an access pattern similar to the above formula, but with concurrent traversals of the inputs, and a single write to the output:

$$\begin{aligned} \widehat{L}_{\text{all}} &= \textsf{Region}\left(|L|\sigma_L \times \left[\sum_{\texttt{r}\in L} \texttt{r.width}\right]\right) \\ \text{PMove}^{*}(X) &= \text{PScan}(X) \odot \text{PWbMv}(X_{\text{all}}) \\ \text{PMove}(L) &= \left[\bigodot_{\texttt{r}\in L^{\text{post}}} \underset{\textsf{R}}{\textsf{S\_Tra\_CR}}\,(\texttt{r}, \sigma_L)\right] \odot \text{PMove}^{*}(L) \\ \text{PMove}(P) &= \left[\bigodot_{\texttt{r}\in P^{\text{post}}} \underset{\textsf{R}}{\textsf{S\_Tra}}\,(\texttt{r})\right] \odot \text{PMove}^{*}(P) \end{aligned}$$

Joining

Joining consists itself of three components: *build*, *probe*, and *materialize*. In the first, a hash table Htbl of length $|P|/2^m$ is constructed whose elements consist either of `p_partkey` values and accompanying pointers, or materialized values of P's attributes, depending on the materialization scheme. In each case:

$$\begin{aligned}
\text{Htbl} &= \mathsf{Region}\left(\frac{|P|}{2^m} \times \mathit{EntrySize}\right) \\
\text{Join} &= \text{JBuild} \odot (\text{JProbe} \odot \text{JMat})
\end{aligned}$$

We consider each plan separately.

No-part When no partitioning is performed, the build and probe phases are:

$$\begin{aligned}
\text{JBuild} &= \underset{\mathsf{R}}{\mathsf{S\_Tra}}\,(P_{\text{key}}) \odot \underset{\mathsf{WA}}{\mathsf{R\_Tra}}\,(\text{Htbl}) \\
\text{JPre} &= \left[\bigodot_{\mathtt{r}\in L^{\text{pre}}} \underset{\mathsf{R}}{\mathsf{S\_Tra}}\,(\mathtt{r})\right] \odot \underset{\mathsf{R}}{\mathsf{S\_Tra\_CR}}\,(L_{\text{key}}, \sigma_L) \\
\text{JProbe} &= \text{JPre} \odot \underset{\mathsf{R}}{\mathsf{R\_Acc}}\,(\text{Htbl}, |L|\sigma_L) \\
\text{JMat}(L) &= \bigodot_{\mathtt{r}\in L^{\text{post}}} \left[\underset{\mathsf{R}}{\mathsf{S\_Tra\_CR}}\,(\mathtt{r}, \sigma_L)\right] \\
\text{JMat}(P) &= \bigodot_{\mathtt{r}\in P^{\text{post}}} \left[\underset{\mathsf{R}}{\mathsf{R\_Acc}}\,(\mathtt{r}, |L|\sigma_L)\right] \\
\text{JMat} &= \text{JMat}(L) \odot \text{JMat}(P) \odot \underset{\mathsf{WA}}{\mathsf{S\_Tra}}\,(\mathtt{Out})
\end{aligned}$$

No-part-mat Even without any partitioning, it is possible during the hash table build phase to materialize the attributes of P so they are instantly available on probes.

$$\begin{aligned}
\text{JBuild} &= \left[\bigodot_{\mathtt{r}\in P^{\text{post}}} \underset{\mathsf{R}}{\mathsf{S\_Tra}}\,(\mathtt{r})\right] \odot \underset{\mathsf{R}}{\mathsf{S\_Tra}}\,(P_{\text{key}}) \odot \underset{\mathsf{WA}}{\mathsf{R\_Tra}}\,(\text{Htbl}) \\
\text{JProbe} &= \text{JPre} \odot \underset{\mathsf{R}}{\mathsf{R\_Acc}}\,(\text{Htbl}, |L|\sigma_L) \\
\text{JMat} &= \bigodot_{\mathtt{r}\in L^{\text{post}}} \left[\underset{\mathsf{R}}{\mathsf{S\_Tra\_CR}}\,(\mathtt{r}, \sigma_L)\right] \odot \underset{\mathsf{WA}}{\mathsf{S\_Tra}}\,(\mathtt{Out})
\end{aligned}$$

Part-single-attr Partitioning on the single (`partkey`) attribute changes the above patterns to reflect division of the hash table into 2^m segments, and the materialization accesses to non-`partkey` attributes to be random.

$$\begin{aligned}
\text{JBuild} &= \bigoplus_{i<2^m} \left[\underset{\mathsf{R}}{\mathsf{S\_Tra}}\left(\widehat{P}_{\text{key}_i}\right) \odot \underset{\mathsf{WA}}{\mathsf{R\_Tra}}\,(\text{Htbl})\right] \\
\text{JProbe} &= \underset{\mathsf{R}}{\mathsf{S\_Tra}}\left(\widehat{L}_{\text{key}}\right) \odot \underset{\mathsf{R}}{\mathsf{R\_Acc}}\,(\text{Htbl}, |L|\sigma_L) \\
\text{JMat}(X) &= \bigodot_{\mathtt{r}\in X^{\text{post}}} \left[\underset{\mathsf{R}}{\mathsf{R\_Acc}}\,(\mathtt{r}, |L|\sigma_L)\right] \\
\text{JMat} &= \text{JMat}(P) \odot \text{JMat}(L) \odot \underset{\mathsf{WA}}{\mathsf{S\_Tra}}\,(\mathtt{Out})
\end{aligned}$$

Part-all-late When all attributes are partitioned along with `partkey`, random materialization accesses become constrained to the span of a partition:

$$\begin{aligned}
\text{JBuild} &= \bigoplus_{i<2^m} \left[\underset{\mathsf{R}}{\mathsf{S\_Tra}} \left(P_{\text{key}_i} \right) \odot \underset{\mathsf{WA}}{\mathsf{R\_Tra}} \left(\text{Htbl} \right) \right] \\
\text{JProbe} &= \underset{\mathsf{R}}{\mathsf{S\_Tra}} \left(\widehat{L}_{\text{key}} \right) \odot \underset{\mathsf{R}}{\mathsf{R\_Acc}} \left(\text{Htbl}, |L|\sigma_L \right) \\
\text{JMat}(P) &= \bigodot_{\mathtt{r}\in\widehat{P}^{\text{post}}} \left[\underset{\mathsf{R}}{\mathsf{R\_Acc}} \left(\mathtt{r}, |L|\sigma_L \right) \right] \\
\text{JMat}(L) &= \bigodot_{\mathtt{r}\in\widehat{L}^{\text{post}}} \left[\underset{\mathsf{R}}{\mathsf{S\_Tra}} \left(\mathtt{r} \right) \right] \\
\text{JMat} &= \bigoplus_{i<2^m} \left[\text{JMat}(\widehat{P}_i) \odot \text{JMat}(\widehat{L}_i) \odot \underset{\mathsf{WA}}{\mathsf{S\_Tra}} \left(\mathtt{Out}_i \right) \right]
\end{aligned}$$

Part-all-early With early materialization, the build and probe phases once more resemble those of the no-part strategy:

$$\begin{aligned}
\text{JBuild} &= \bigoplus_{i<2^m} \left[\underset{\mathsf{R}}{\mathsf{S\_Tra}} \left(\widehat{P}_i \right) \odot \underset{\mathsf{WA}}{\mathsf{R\_Tra}} \left(\text{Htbl} \right) \right] \\
\text{JProbe} &= \bigoplus_{i<2^m} \left[\underset{\mathsf{R}}{\mathsf{S\_Tra}} \left(\widehat{L}_i \right) \odot \underset{\mathsf{R}}{\mathsf{R\_Acc}} \left(\text{Htbl}, |L|\sigma_L \right) \right] \\
\text{JMat} &= \underset{\mathsf{W}}{\mathsf{S\_Tra}} \left(\mathtt{Out} \right)
\end{aligned}$$

7.6 Proof of NP-Hardness of Single-Phase Allocation

Theorem 6. *Single-phase allocation is NP-hard in the number of regions.*

Proof. To see this, it suffices to show that the allocation of space between multiple repeated, unidirectional sequential traversals can effectively solve the knapsack problem. The latter is, as formulated in Rajkumar [54], the set $a_i \in \{0, 1\}$ which maximizes:

$$\begin{aligned}
\text{Maximize: } & \underset{a_1,\ldots,a_n}{\arg\max} \sum_{i=1}^{n} a_i v_i \\
\text{Subject to: } & \sum_{i=1}^{n} a_i R_i \leq \mathbf{R}
\end{aligned} \tag{7.32}$$

Assume there are n resources, of which the ith costs R_i and contributes value v_i, and that at most $\mathbf{R}$ cost can be spent.

Now, consider n regions $\mathtt{R}_i$, of size $|\mathtt{R}_i|$, and n repetitive sequential traversals:

$$\mathtt{P} = \bigodot_{i\leq n} \mathsf{RS\_Tra} \left(N_i, \mathtt{R}_i, \mathbf{dir} = \mathbf{uni} \right)$$

Under the Manegold model, an optimal allocation of cache $\mathcal{C}$ is one that *minimizes*

$$\begin{aligned} \text{Minimize: } & \sum_{i\leq n} |\mathtt{R}_i|_{\mathcal{C}} + \sum_{i\leq n}(1-a_i)(N_i-1)|\mathtt{R}_i| \\ a_i &= \begin{cases} 0 & \mathtt{A}_i < |\mathtt{R}_i| \\ 1 & \mathtt{A}_i \geq |\mathtt{R}_i| \end{cases} \\ \text{Subject to: } & \sum_{i\leq n} \mathtt{A}_i = \sum_{i\leq n} a_i|\mathtt{R}_i| \leq |\mathcal{C}| \end{aligned} \tag{7.33}$$

It is clear, however, that a solution $a^* = (a_1, \ldots, a_n)$ which minimizes the above objective must also *maximize*

$$\sum_{i\leq n} a_i[(N_i-1)|\mathtt{R}_i|]$$

We can convert this into (7.32) by setting each repetition count to

$$N_i = 1 + \frac{v_i}{|\mathtt{R}_i|_{\mathcal{C}}}$$

And so solve the 0-1 inexact knapsack problem. □ □

7.7 2x Error Bound on Convex Hull Allocation

In order to attain polynomial-time single-phase allocation, the outer loop of Algorithm 7 may be skipped if each cost function is replaced by its convex hull, thereby permitting direct application of Katoh et al.'s RAP algorithm [24].

As shown by Rajkumar et al. [54], the deviation from optimal cost that results from the approximation is bounded by the maximum distance between any individual utility function and its convex hull. For the cost functions composing the Manegold model and our extensions, this is bounded by the parameters depicted in Figure 7.4. In each case, the maximum traffic is bounded from below by the footprint $\mathbf{F}_{\mathcal{C}}(\mathtt{P})$, and from above *number of accesses* it makes. Assuming the discontinuity removal optimization of Section 7.2, the convex threshold point will occur at half the footprint size, which is half the relation size in the example of Figure 7.4. That is also the point of maximum distance between hull and the cost function, which can be expressed as

$$\delta \leq \frac{1}{2}\left(||\mathtt{R}|| - \mathbf{F}_{\mathcal{C}}(\mathtt{P})\right) \leq \frac{1}{2}||\mathtt{R}||$$

In other words, the error δ is at most half of the true value $||R||$ at the convex threshold.

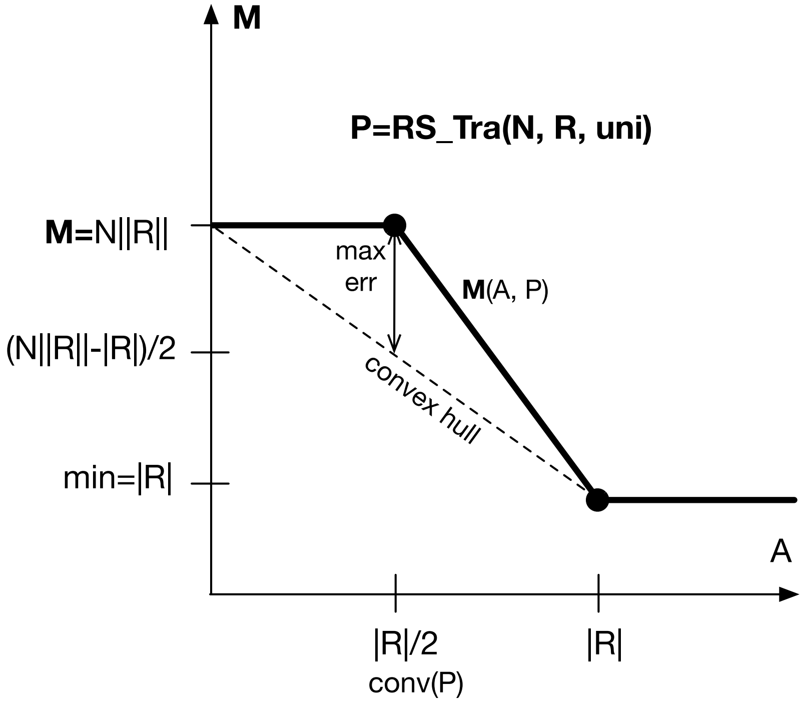


Figure 7.4: Convex Hull of Non-Convex (Fig 7.3b) Cost Function

Chapter 8

Conclusions & Future Work

8.1 Summary

This thesis makes the bet that the performance of database analytics workloads either is or will soon be limited by the memory bandwidth bottleneck, and so future hardware to accelerate them, if it is indeed worthwhile, should be designed to reduce communication requirements, or eliminate computational bottlenecks that prevent communication-minimizing plans from saturating existing bandwidth capacities. Put another way, it wagers that future hardware should be designed in tandem with changes to query plans that account for the scarcity of this resource (Chapter 1).

Accordingly, the concept of *machine-level plans* is introduced to encompass all the lowest-level aspects of query optimization impacting the interaction of query execution with machine microarchitecture, and which many prior researchers have considered extensively at the level of individual relational operators, such as joins. The impacts of machine-level choices are considered at the level of fully-compiled *queries*, rather than operators, and classified in a taxonomy that describes their effects in terms of communication and other hardware-related constraints of machines that exist today (Chapter 2). Since the space of machine-level plans for individual queries is potentially large, and different choices can result in quite different code structures (i.e. loop nests), a new abstraction in the form of the HiRes language is presented to represent them concisely, and to facilitate the investigation of tradeoffs they present.

Several challenges and opportunities arise in the process of compiling HiRes programs. The semantics of relational operators in query plans more generally, and of low-level HiRes primitives in particular, make *loop fusion* an under-constrained optimization problem to which the space of possible solutions is difficult to express intuitively, but whose impact on performance is potentially large (Chapter 4). Meanwhile, the process of refining HiRes to C++ motivates the introduction of an intermediate representation, LoRes, on which type-sensitive transformation passes can be perfomed to achieve high performance while also simplifying the architecture of the HiRes compiler frontend (Chapter 5).

8.2 Principal Findings & Recommendations

An initial evaluation of compiled HiRes query plans on contemporary server-grade, general-purpose CPUs reveals that, with much effort and a broad search of machine-level plan spaces, even complex queries can achieve close to 35-50% of the peak theoretical performance that (essentially) any plan could attain on platforms with equivalent bandwidth, raising questions about the degree to which future hardware might be able to accelerate them by more than 2-3x.

This suggests that if future performance advances do come from hardware improvements, they will likely result from changes to support more advanced compression techniques, the disaggregation of many-core CPUs into greater numbers of sockets, or on-die and on-package memory technologies. In the case of the latter, though, it will need to be shown that sufficient locality can be found to exploit them; none of the test queries examined in this study would have benefited from HBM by more than 20%.

At the same time, the preponderance of plans running an order of magnitude slower than the best found for a given query means that performance cliffs are abundant and often difficult to avoid. A key challenge for query optimizers of the future will be to exploit the opportunities of machine-level plan choices while avoiding their inherent pitfalls.

8.3 Limitations & Future Work

The above conclusions should be taken as extremely provisional, on account of the limited scope of study, and hardware-conscious optimizations that still cannot be expressed in the HiRes language. It is quite possible that the expansion of coverage to ever more complex queries and plans will reveal ample opportunity for acceleration. Regrettably, the construction of plan generators for even simple queries has proven cumbersome, suggesting inadequacies in the HiRes language's design that could someday be improved.

This fragility means that the construction of good query optimizers for HiRes or similar plan formats will pose challenges beyond those already considered difficult in the database community. Indeed, the lack of a SQL front end and automated optimizer in Ressort is a fundamental limitation of the foregoing results. The introduction of slightly higher-level constructs in the form of operator macros (Chapter 3 Section 3.3) represents one small step towards the construction of an algebrizer, but this remains an unfinished project.

In order to approach competitive performance on today's general-purpose platforms, several compiler and language features unrelated to pure communication costs were needed, and still others could improve empirical results further. For example, SIMD or vector instructions are known to be necessary to reach peak throughput for some physical operators (such as filtering) on machines like those used in this study [48]), but Ressort cannot yet generate them. The NUMA-aware partitioning strategy of Schuh et al. [59]–which entails merging per-thread partition buffers during the join phase–cannot be fully expressed due to the lack of an appropriate merge operation. Software prefetching of randomly-accessed data could mitigate the pipeline stalls associated with the cache misses they engender, especially as part of cracked attribute accesses, or during post-join attribute gathers. Additionally, support for data-flow abstractions that enable the accumulation of fixed-size blocks of surviving tuples across selective filters or joins before processing by a downstream operator could allow better use of existing bandwidth capacity. At present, neither these strategies, nor even conditional branch-avoidance techniques inside of the `Collect()` operator, can be expressed.

These caveats apply even before considering more specialized hardware platforms already in wide use, such as GPUs, which did not enter into this study. Since the primary motivation of Ressort was to enable both the design of and optimization of database software for future hardware, the extension of code generation abilities to existing data-parallel programming models is a natural next step, and it should be hoped that the same holds for whatever more specialized hardware architectures ultimately emerge. If HiRes is truly to contribute to their design, however, then the most immediate pursuit should be the automation of Manegold-style [38] access pattern algebra (Chapter 7) generation and communication cost analysis as part of the HiRes compilation process.

8.4 A Methodological Plea

Until the advent and widespread adoption of bandwidth-expanding technologies such as on-chip silicon photonics, bandwidth to DRAM will continue to represent an absolute limit on query processing performance, whether or not existing software and hardware systems manage to exhaust it. Accordingly, it should be hoped that future research results in both the hardware and software domains be judged relative to that limit. By framing results in terms of this upper bound, researchers will be able to state more readily how much work remains to be done, and how much (or little) performance remains to be had for these workloads.

Of course, such analysis will not always be straightforward. For complex queries, the full space of possible machine-level plans (assuming an optimal logical plan choice, which itself may be unrealistic) will have to be evaluated relative to a given memory hierarchy, and the bandwidth limit will have to be established by considering the least costly of these in communication terms. But HiRes contributes to the realization of that goal by providing a higher level of abstraction for encoding the relevant plan choices and exploring the search space, and we hope that future development of it, or similar query compilation frameworks, will eventually permit researchers to build a shared understanding of communication bounds for common benchmark queries.

Bibliography

[1] Sandeep R Agrawal et al. "A many-core architecture for in-memory data processing". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2017, pp. 245–258.

[2] Alfred V Aho et al. *Compilers: Principles, Technologies, and Tools*. 2006.

[3] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. "Massively parallel sort-merge joins in main memory multi-core database systems". In: *Proceedings of the VLDB Endowment* 5.10 (2012), pp. 1064–1075.

[4] Joy Arulraj and Andrew Pavlo. "How to build a non-volatile memory database management system". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. 2017, pp. 1753–1758.

[5] Cagri Balkesen et al. "Multi-core, main-memory joins: Sort vs. hash revisited". In: *Proceedings of the VLDB Endowment* 7.1 (2013), pp. 85–96.

[6] Guy E. Blelloch and John Greiner. "A Provable Time and Space Efficient Implementation of NESL". In: *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*. ICFP '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 213–225. ISBN: 0-89791-770-7. DOI: `10.1145/232627.232650`. URL: `http://doi.acm.org/10.1145/232627.232650`.

[7] Peter A Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *CIDR*. Vol. 5. 2005, pp. 225–237.

[8] Sebastian Breß, Henning Funke, and Jens Teubner. "Robust query processing in co-processor-accelerated databases". In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 1891–1906.

[9] Sebastian Breß et al. "Generating custom code for efficient query execution on heterogeneous processors". In: *The VLDB Journal-The International Journal on Very Large Data Bases* 27.6 (2018), pp. 797–822.

[10] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[11] Leonardo Dagum and Ramesh Menon. “OpenMP: An Industry-Standard API for Shared-Memory Programming”. In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1070-9924. DOI: `10.1109/99.660313`. URL: `https://doi.org/10.1109/99.660313`.

[12] Martin Dietzfelbinger. “Universal hashing and k-wise independent random variables via integer arithmetic without primes”. In: *STACS 96*. Ed. by Claude Puech and Rüdiger Reischuk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 567–580.

[13] Jaeyoung Do et al. “Query processing on smart SSDs: opportunities and challenges”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 1221–1230.

[14] Freee Software Foundation, Inc. *GCC, The GNU Compiler Collection*. `https://gcc.gnu.org`.

[15] G Gao et al. “Collective loop fusion for array contraction”. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 1992, pp. 281–295.

[16] G. Graefe. “Volcano/spl minus/an extensible and parallel query evaluation system”. In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (Feb. 1994), pp. 120–135. ISSN: 1041-4347. DOI: `10.1109/69.273032`.

[17] Timothy Hayes et al. “Vector extensions for decision support DBMS acceleration”. In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2012, pp. 166–176.

[18] Timothy Hayes et al. “VSR sort: A novel vectorised sorting algorithm & architecture extensions for future microprocessors”. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 26–38.

[19] Intel Corp. *Product Brief: Intel Optane DC Persistent Memory*. `https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf`. (Visited on 10/29/2019).

[20] Intel Corp. *The SAP Business ByDesign Solution Powered by Intel Xeon Scalable Processors*. `https://www.intel.com/content/dam/www/public/us/en/documents/brief/cloud-based-erp-white-paper.pdf`. (Visited on 11/24/2016).

[21] John C. McCallum. *Memory Prices (1957-2017)*. Published at ”`http://www.jcmit.com/memoryprice.htm`”. 2017.

[22] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2017, pp. 1–12.

[23] Kaan Kara, Jana Giceva, and Gustavo Alonso. “Fpga-based data partitioning”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. 2017, pp. 433–445.

[24] N Katoh, T Ibaraki, and H Mine. "A polynomial time algorithm for the resource allocation problem with a convex objective function". In: *Journal of the Operational Research Society* 30.5 (1979), pp. 449–455.

[25] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots". In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE. 2011, pp. 195–206.

[26] Changkyu Kim et al. "Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs". In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1378–1389.

[27] Onur Kocberber et al. "Meet the Walkers". In: *Proceedings of the International Symposium on Microarchitecture (MICRO)*. Dec. 2013.

[28] Andrew Lamb et al. "The vertica analytic database: C-store 7 years later". In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1790–1801.

[29] Harald Lang et al. "Massively parallel NUMA-aware hash joins". In: *In Memory Data Management and Analysis*. Springer, 2015, pp. 3–14.

[30] Glen G Langdon Jr. "Database Machines: An Introduction." In: *IEEE Transactions on Computers* 28.6 (1979), pp. 381–383.

[31] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[32] Chen Lee et al. "On quality of service optimization with discrete QoS options". In: *Real-Time Technology and Applications Symposium, 1999. Proceedings of the Fifth IEEE*. IEEE. 1999, pp. 276–286.

[33] Viktor Leis et al. "How good are query optimizers, really?" In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 204–215.

[34] Yinan Li and Jignesh M Patel. "BitWeaving: fast scans for main memory data processing". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 289–300.

[35] Eric Love. "Ressort: An Auto-Tuning Framework for Parallel Shuffle Kernels". In: (2015).

[36] Jason Lowe-Power, Mark D. Hill, and David A. Wood. "When to use 3D Die-Stacked Memory for Bandwidth-Constrained Big Data Workloads". In: *CoRR* abs/1608.07485 (2016). arXiv: `1608.07485`. URL: `http://arxiv.org/abs/1608.07485`.

[37] Stefan Manegold, Peter Boncz, and Martin Kersten. "Optimizing main-memory join on modern hardware". In: *IEEE Transactions on Knowledge and Data Engineering* 14.4 (2002), pp. 709–730.

[38] Stefan Manegold, Peter Boncz, and Martin L Kersten. "Generic database cost models for hierarchical memory systems". In: *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment. 2002, pp. 191–202.

[39] McCalpin, John D. *Memory Bandwidth and System Balance in HPC Systems*. `https://sites.utexas.edu/jdm4372/2016/11/22/sc16-invited-talk-memory-bandwidth-and-system-balance-in-hpc-systems/`. (Visited on 11/22/2016).

[40] Nimrod Megiddo and Vivek Sarkar. "Optimal weighted loop fusion for parallel programs". In: *SPAA*. Vol. 97. Citeseer. 1997, pp. 282–291.

[41] Gordon E Moore. *Cramming more components onto integrated circuits*. 1965.

[42] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. École Polytechnique Fédérale de Lausanne, 2004.

[43] Oracle. *Oracle's SPARC T7 and SPARC M7 Server Architecture*. Published at "`http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/sparc-t7-m7-server-architecture-2702877.pdf`". 2016.

[44] Shoumik Palkar et al. "Evaluating End-to-end Optimization for Data Analytics Applications in Weld". In: *Proc. VLDB Endow.* 11.9 (May 2018), pp. 1002–1015. ISSN: 2150-8097. DOI: `10.14778/3213880.3213890`. URL: `https://doi.org/10.14778/3213880.3213890`.

[45] Holger Pirk et al. "CPU and cache efficient management of memory-resident databases". In: *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE. 2013, pp. 14–25.

[46] Holger Pirk et al. "Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware". In: *Proc. VLDB Endow.* 9.14 (Oct. 2016), pp. 1707–1718. ISSN: 2150-8097. DOI: `10.14778/3007328.3007336`. URL: `http://dx.doi.org/10.14778/3007328.3007336`.

[47] Constantin Pohl, Kai-Uwe Sattler, and Goetz Graefe. "Joins on high-bandwidth memory: a new level in the memory hierarchy". In: *The VLDB Journal* (2019), pp. 1–21.

[48] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. "Rethinking SIMD vectorization for in-memory databases". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1493–1508.

[49] Orestis Polychroniou and Kenneth A Ross. "A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 755–766.

[50] Orestis Polychroniou and Kenneth A Ross. "Efficient lightweight compression alongside fast scans". In: *Proceedings of the 11th International Workshop on Data Management on New Hardware*. ACM. 2015, p. 9.

[51] Orestis Polychroniou and Kenneth A Ross. “High throughput heavy hitter aggregation for modern SIMD processors”. In: *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. ACM. 2013, p. 6.

[52] Jason Power et al. “Toward GPUs being mainstream in analytic processing: An initial argument using simple scan-aggregate queries”. In: *Proceedings of the 11th International Workshop on Data Management on New Hardware*. ACM. 2015, p. 11.

[53] Georgios Psaropoulos et al. “Bridging the latency gap between NVM and DRAM for latency-bound operations”. In: *Proceedings of the 15th International Workshop on Data Management on New Hardware*. CONF. ACM. 2019.

[54] Ragunathan Rajkumar et al. “Practical solutions for QoS-based resource allocation problems”. In: *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*. IEEE. 1998, pp. 296–306.

[55] Stefan Richter, Victor Alvarez, and Jens Dittrich. “A Seven-dimensional Analysis of Hashing Methods and Its Implications on Query Processing”. In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 96–107. ISSN: 2150-8097. DOI: 10.14778/2850583.2850585. URL: http://dx.doi.org/10.14778/2850583.2850585.

[56] Samsung Electronics Co., Ltd. *Samsung Electronics Introduces New High Bandwidth Memory Technology Tailored to Data Centers, Graphic Applications, and AI*. https://www.samsung.com/semiconductor/insights/tech-leadership/samsung-electronics-introduces-new-high-bandwidth-memory-technology-tailored-to-data-centers-graphic-applications-and-ai/. (Visited on 10/31/2019).

[57] Nadathur Satish et al. “Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 351–362.

[58] Tao B Schardl, William S Moses, and Charles E Leiserson. “Tapir: Embedding fork-join parallelism into LLVM’s intermediate representation”. In: *ACM SIGPLAN Notices*. Vol. 52. 8. ACM. 2017, pp. 249–265.

[59] Stefan Schuh, Xiao Chen, and Jens Dittrich. “An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016, pp. 1961–1976. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882917. URL: http://doi.acm.org/10.1145/2882903.2882917.

[60] Amir Shaikhha et al. “How to Architect a Query Compiler”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016, pp. 1907–1922. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2915244. URL: http://doi.acm.org/10.1145/2882903.2915244.

[61] Minglong Shao, Anastassia Ailamaki, and Babak Falsafi. "DBmbench: fast and accurate database workload representation on modern microarchitecture". In: *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON).* 2005.

[62] Robert M Shapiro and Harry Saint. *The representation of algorithms*. Tech. rep. Applied Data Research Inc. New York Corporate Research Center, 1969.

[63] Lefteris Sidirourgos and Martin Kersten. "Column imprints: a secondary index structure". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 893–904.

[64] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. "Vectorization vs. compilation in query execution". In: *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. ACM. 2011, pp. 33–40.

[65] Harold S. Stone, John Turek, and Joel L. Wolf. "Optimal partitioning of cache memory". In: *IEEE Transactions on computers* 41.9 (1992), pp. 1054–1068.

[66] Mike Stonebraker et al. "C-store: a column-oriented DBMS". In: *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 2005, pp. 553–564.

[67] Tableau. *Tableau Business Intelligence and Analytics Software*. `https://www.tableau.com`. (Visited on 10/28/2019).

[68] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. "How to Architect a Query Compiler, Revisited". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 307–322. ISBN: 978-1-4503-4703-7. DOI: `10.1145/3183713.3196893`. URL: `http://doi.acm.org/10.1145/3183713.3196893`.

[69] The United States Census Bureau. *The Hollerith Machine : 1888 Competition*. `https://www.census.gov/history/www/innovations/technology/the_hollerith_tabulator.html`. (Visited on 10/24/2019).

[70] Transaction Processing Performance Council. *TPC-H benchmark specification*. 2014.

[71] Thomas Willhalm et al. "Vectorizing Database Column Scans with Complex Predicates." In: *ADMS@ VLDB*. 2013, pp. 1–12.

[72] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[73] Lisa Wu et al. "Navigating big data with high-throughput, energy-efficient data partitioning". In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 249–260.

[74] Lisa Wu et al. "Q100: the architecture and design of a database processing unit". In: *ACM SIGPLAN Notices* 49.4 (2014), pp. 255–268.

[75] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. “The Yin and Yang of processing data warehousing queries on GPU devices”. In: *Proceedings of the VLDB Endowment* 6.10 (2013), pp. 817–828.

[76] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. “Vectorwise: A vectorized analytical DBMS”. In: *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE. 2012, pp. 1349–1350.