Sharing Aggregate Computation for Distributed Queries

Ryan Jay Huebsch
Minos Garofalakis
Joseph M. Hellerstein
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

July 12, 2006

Sharing Aggregate Computation for Distributed Queries

Ryan Huebsch, Minos Garofalakis, Joseph M. Hellerstein, and Ion Stoica
University of California at Berkeley & Intel Research Berkeley

Abstract

An important challenge in modern distributed querying is to efficiently process multiple continuous aggregation queries simultaneously. However, processing each of these queries independently may be prohibitive due to network bandwidth constraints. Multiquery optimizations can be used to share computations across queries in order to reduce the overall network communication. In this paper, we consider this problem in the context of distributed aggregation queries that vary in their selection predicates. We identify settings in which a large set of n such queries can be answered by executing $k << n$ *different* queries. The k queries are revealed by analyzing a boolean matrix capturing the connection between data and the queries that they satisfy, in a manner akin to familiar techniques from linear algebra. Indeed, we identify a class of *linear* aggregate functions (including SUM, COUNT and AVERAGE), and show that the sharing potential for such queries can be optimally recovered using standard matrix decompositions from computational linear algebra. Unfortunately, for some other typical aggregation functions (including MIN and MAX) we find that optimal sharing maps to the NP-hard *set basis* problem. However, for those scenarios, we present a family of heuristic algorithms that perform well for moderately-sized matrices. We also present an overall distributed system architecture to exploit sharing opportunities, and experimentally evaluate the benefits of our techniques via a novel, flexible random workload generator we develop for this setting.

1 Introduction

There is a large and growing body of work on the design of distributed query systems. The focus of much of this work has been on the efficient execution of individual, one-shot queries, through intelligent data-processing algorithms, data/query shipping strategies, etc. Recent years, however, have witnessed the emergence of a new class of *large-scale distributed monitoring* applications – including network-traffic monitors, sensornets, and financial data trackers – that pose novel data-management challenges. First, many monitoring tasks demand support for *continuous queries* instead of ad-hoc requests, to accurately track the current state of the environment being monitored. Second, given the inherently distributed nature of such systems, it is crucial to minimize the *communication overhead* that monitoring imposes on the underlying infrastructure, e.g., to limit the burden on the production network [5] or to maximize sensor battery life [16].

In most monitoring scenarios, the naive warehousing "solution" of simply collecting a large, distributed data set at a centralized site for query processing and result dissemination is prohibitively expensive in terms of both latency and communication cost – and often, simply unnecessary. The amount of data involved can be so large and dynamic in nature that it can easily overwhelm typical users or applications with too much detailed information. Instead, high-level, *continuous aggregation queries* are routinely employed to provide meaningful summary information on the underlying distributed data collection and, at the same time, to allow users to iteratively drill-down to interesting regions of the data. Typical aggregation queries also allow for effective, *in-network processing* that can drastically reduce communication overheads by "pushing" the aggregate function computation down to individual nodes in the network [14].

Another crucial requirement for large-scale distributed monitoring platforms is the ability to *scale* in both the volume of the underlying data streams and the number of simultaneous long-running queries. As an example, consider the Network Operations Center (NOC) for the IP-backbone network of a large ISP (such as Sprint or AT&T). Such NOCs routinely need to track (in real time) hundreds of continuous queries collecting aggregate statistics over thousands of network elements (routers, switches, links, etc.) and extremely high-rate event streams at different layers of the network infrastructure. This requirement emphasizes a new class of multi-query optimization problems that focus on dynamically sharing execution costs across continuous stream queries to optimize overall system performance.

Our Contributions. In this paper, we focus on dynamic multi-query optimization techniques for continuous aggregation queries over physically distributed data streams. In a nutshell, we demonstrate opportunities to compute q aggregation queries that vary only in their selection predicates via $k << q$ queries that may, in fact, be *different* from the input queries themselves. Our method is to formalize the shared optimization problem through the analysis of a dynamic *fragment matrix* that captures the connection between observed data and query predicates, and the algebraic properties of the underlying aggregate function. This leads us to algorithmic solutions for these problems grounded in linear

algebra and novel combinatorial optimization techniques. More concretely, our main contributions can be summarized as follows.

• **Algebraic Query/Data Decomposition.** We identify sharing opportunities across different aggregation functions through the analysis of a dynamic, *boolean fragment matrix* that accurately captures the disjoint fragments of the streaming data tuples with respect to the underlying selection predicates. The basic intuition here is that the set of computed aggregates can be effectively compressed by *decomposing* the fragment matrix into *"independent components"*, which are sufficient to reconstruct every individual aggregate answer. The exact notion of an "independent component" varies depending on the algebraic characteristics of the underlying aggregate (e.g., linear or duplicate-insensitive), resulting in optimization problems of varying computational complexity.

• **Novel Optimization Algorithms for Distributed Aggregate Sharing.** Based on our insights from the fragment-matrix model, we formulate our sharing optimization problem for distributed aggregates for different classes of aggregation functions. For the class of *linear* aggregates (e.g., COUNT, SUM) we show efficient, optimal sharing strategies based on existing *linear-algebra techniques* (such as LU, QR, or SVD decompositions). Unfortunately, duplicate-insensitive aggregates (e.g., MIN, MAX) result in a dramatic increase in problem complexity, since the problem maps to the NP-hard *Set-Basis Problem* [7] (known to be inapproximable to within any constant factor [13]); thus, we propose a novel efficient heuristic technique that, as our empirical results demonstrate, performs well in practice. We also give an analysis of the sharing benefits.

• **Extensive Experimental Results Validating our Approach.** We develop a flexible workload generator for our problem that allows us to explicitly and flexibly control the degree of benefit available in the workload. Our experimental results clearly demonstrate that our algorithms can provide dramatic communication savings. For linear aggregate functions, two of the theoretically-optimal methods achieve 100% of the potential benefit under all settings, despite the potential for numerical instability in floating point computations; instabilities in the third technique reduce its effectiveness in some cases. For duplicate insensitive aggregates, the best of our methods approaches 90% of optimal across a wide range of workloads.

Prior Work. For the case of a *single* distributed aggregation query, efficient in-network execution strategies have been proposed by several recent papers and research prototypes (including, for instance, TAG [14], SDIMS [19], and PIER [9]). The key idea in these techniques is to perform the aggregate computation over a dynamic tree in an overlay network. Aggregation occurs over a dynamic tree, with each node combining the data found locally along with any *Partial State Records (PSRs)* it receives from its children, and forwarding the resulting PSR one hop up the tree.

Over time, the tree dynamically adjusts to changing node membership and network conditions. More recent work on distributed data streaming has demonstrated that, with appropriate PSR definitions and combination techniques, in-network aggregation ideas can be extended to fairly complex aggregates, such as approximate quantiles [4, 8], and approximate histograms and join aggregates [3]. None of this earlier work considers the case of multiple distributed aggregation queries, essentially assuming that such queries are processed individually, modulo perhaps some simple routing optimizations. For example, PIER suggests using distinct routing trees for each query in the system, in order to balance the network load [9].

In the presence of hundreds or thousands of continuous aggregation queries, system performance and scalability depend upon effective sharing of execution costs across queries. Recent work has suggested solutions for the *centralized* version of the problem, where the goal is to minimize the amount of computation involved when tracking (1) several GROUP-BY aggregates (differing in their grouping attributes) [20], or (2) several windowed aggregates (differing in their window sizes and/or selection predicates) [10, 11], over a continuous data stream *observed at a single site*. In the distributed setting, network communication is the typical bottleneck, and hence communication overheads become an important optimization concern.

In an independent effort, [18] proposes a distributed solution for linear aggregates. Their scheme is based on heuristics tailored to power-constrained sensornets where the query workload is restricted to a *static* collection of simple spatial predicates related to the network topology. Instead, our dynamic fragment-based method does not have any restrictions on the query predicates, and employs optimal linear-algebra techniques to uncover sharing across linear aggregates. They also observe the analogy to the Set-Basis problem for MIN/MAX aggregates but do not propose any algorithmic solution for the duplicate-insensitive case.

2 Overview

The goal of the algorithms we present in this paper is to minimize overall network communication. During an aggregation query, each node must send a partial state record (PSR) to its parent in an aggregation tree. If there is no sharing, then we are communicating one partial state record (PSR) per node per query per window. If we have q queries, our goal is to only send k PSRs per node per window, where $k << q$, such that the k PSRs are sufficient to compute the answer to all q queries. The next section discusses the intuition for how to select these k PSRs.

2.1 The Intuition

Consider a very simple distributed monitoring example system with n nodes. Each of the nodes are examining their

local stream of packets. Each packet is annotated with three boolean values: (1) whether there is a reverse DNS entry for the source, (2) if the source is on a spam blacklist, and (3) if the packet is marked suspicious by an intrusion detection system (IDS). One could imagine various applications monitoring all n streams at once by issuing a continuous query to count the number of global "bad" packets, where each application determines "bad" as some predicate over the three flags. Here are example query predicates from five COUNT queries over the stream of packets from all the nodes:

1. `WHERE noDNS = TRUE`
2. `WHERE suspicious = TRUE`
3. `WHERE noDNS = TRUE OR suspicious = TRUE`
4. `WHERE onSpamBlackList = TRUE`
5. `WHERE onSpamBlackList = TRUE`
   `AND suspicious = TRUE`

We will use an idea from Krishnamurthy, et al. [10] to get an insight for how to execute these queries using fewer than 5 PSRs. In their work, they look at the set of queries that each tuple in the stream satisfies, and use this classification to partition the tuple-space to minimize the number of aggregation operations (thereby reducing computation time). Returning to our five example queries above, suppose in a single window at node i we have tuples that can be partitioned into exactly one of the following five categories:

1. Tuples that satisfy queries 1 and 3 only
2. Tuples that satisfy queries 2 and 3 only
3. Tuples that satisfy query 4 only
4. Tuples that satisfy queries 1, 3, and 4 only
5. Tuples that satisfy queries 2, 3, 4 and 5 only

We will refer to each of these categories as a *fragment*. As a compact notation, we can represent this as a $(f \times q)$ boolean *fragment matrix*, F, with each column representing a query (numbered from left to right) and each row representing a fragment:

$$
\begin{array}{c}
\text{Query 1} \downarrow \qquad ... \qquad \downarrow \text{Query 5} \\
F = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{array}{l} \leftarrow \text{Fragment 1} \\ \\ ... \\ \\ \leftarrow \text{Fragment 5} \end{array}
\end{array}
$$

Now, suppose in a given window some node i receives a number of tuples corresponding to each fragment; e.g., it receives 23 tuples satisfying queries 1 and 3 only (row 1), 43 satisfying queries 2 and 3 only (row 2), etc. We can also represent this as a matrix called A_i:

$$
A_i^{\mathsf{T}} = \begin{bmatrix} 23 & 43 & 18 & 109 & 13 \end{bmatrix}
$$

Given the two matrices, we can now compute the local count for the first query (the first column of F) by summing

the first and fourth entries in A_i, the second query by summing the second and fifth entries in A_i. In algebraic form $A_i^{\mathsf{T}} \times F$ will produce a one-row matrix with each column representing the count for the respective query. Encoding the information as matrix A_i is not more compact than sending the traditional set of five PSRs (one for each query). However, if we can find a reduced matrix A_i' – one with empty entries that do not need to be communicated – such that $A_i'^{\mathsf{T}} \times F = A_i^{\mathsf{T}} \times F$, we can save communication at the expense of more computation.

This is indeed possible in our example. First, note that fragment 4 is the OR of fragments 1 and 3 with no overlap (the conjunction equals zero). Now, observe the significance of that fact with respect to computing our COUNT queries: when summing up the counts for those queries that correspond to fragment 1 (queries 1 and 3), we can ignore the count of fragment 3 since its entries for those queries are zero. Similarly, when summing up the counts for queries overlapping fragment 3 (query 4), we can ignore the count of fragment 1. Because of this property, we can add the count associated with fragment 4 into *both* of the counts for fragments 1 and 3 without double-counting in the final answer, as follows:

$$
A'^{\mathsf{T}} = \begin{bmatrix} 23+109=132 & 43 & 18+109=127 & 109 \rightarrow \emptyset & 13 \end{bmatrix}
$$

Using this new A_i', $A_i'^{\mathsf{T}} \times F$ will still produce the correct answer for each query, even though A' has more empty entries. And since A_i' has an empty entry, there is a corresponding savings in network bandwidth, sending only four PSRs instead of five. In essence, we only need to execute four queries instead of the original five. The key observation is that the size of A_i' is equal to the number of *independent rows* in F, or the *rank* of F; the exact definition of independence depends on the aggregation function as we discuss next. In all cases the rank of F will always be less than or equal to $min(f, q)$. Therefore we will never need more than q PSRs, which is no worse than the no-sharing scenario.

2.2 Taxonomy Of Aggregates

The particular optimization presented in the previous section works for all distributive and algebraic aggregate functions. However, some aggregate functions have special properties that allow more powerful solutions to be used that exploit additional sharing opportunities. We categorize aggregates into three broad categories: *linear*, *duplicate insensitive*, and *general*. These three categories map to different variations of the problem and require separate solutions. We first discuss the taxonomy and then briefly introduce our solutions.

Formally, we use the term *linear* for aggregate functions whose fragment matrix entries form a field (in the algebraic sense) under two operations, one used for combining rows, the other for scaling rows by constants. An important necessary property of a field is that there be *inverses* for all values under both operators. Among the familiar SQL

aggregates, note that there is no natural inverse for MIN and MAX under the natural combination operator: given that $z = \text{MAX}(x, y)$, there is no unique y^{-1} such that $\text{MAX}(z, y^{-1}) = x$. Hence these are not linear. Another category we consider are *duplicate insensitive* aggregates, which produce the same result regardless of the number of occurrences of a specific datum. The table below lists a few example aggregate functions for each category:

	Non-linear	Linear
Duplicate Sensitive	k-MAX, k-MIN	SUM, COUNT, AVERAGE
Duplicate Insensitive	MIN, MAX, BLOOM FILTER, logical AND/OR	Spectral Bloom filters [2], Set expressions with updates [6]

The intuition for why k-MAX and k-MIN (the multi-set of the top k highest/lowest datums) are non-linear is analogous to that of MAX and MIN. k-MAX/MIN are also duplicate sensitive since evaluating each additional copy of the same highest datum would expel the kth highest datum due to the multi-set semantics.

Spectral Bloom filters are an extension of Bloom filters that keep a frequency associated with each bit. The frequency is incremented when a datum maps to that bit, and can be decremented when a datum is removed from the filter. This is linear because the frequencies can be added/subtracted to each other and can be scaled by a real number. In addition the output of the filter is based on whether the frequency is greater than zero or not, so counting the same datum twice may produce an inflated frequency value but does not change the output.

In Section 4 we address linear aggregates where this problem can be reduced directly to rank-revealing linear algebra factorization of matrix F, and polynomial-time techniques from the literature directly lead us to an efficient solution. For duplicate insensitive aggregates, we explain in Section 5 that the problem is a known NP-Hard problem and has higher computational complexity; in these cases we develop a family of heuristics that we evaluate experimentally. Finally for aggregates that are neither linear or duplicate insensitive, the most conservative optimization algorithm must be used. We stress that for both linear and duplicate insensitive aggregates, our solutions will never require more global aggregate computations than the no-sharing scenario.

We now discuss the architecture and the general solution to this problem.

2.3 Architecture

The general technique for performing multi-query optimization has four phases. First at each node, i, we need to create the initial F and A_i matrices in the *fragmentation* phase. Second, we can *decompose* F and A_i into a smaller A_i'. Third, we perform the *global aggregation* of all local A_i''s across all nodes. Finally, we can *reconstruct* the final answers to each query at some node j. This process is illus-

trated in Figure 1 and described in detail below.

In the first phase, fragmentation, we are using the same technique presented in [10]. Each tuple is locally evaluated against each query's predicates to determine on-the-fly which fragment the tuple belongs to. We can use techniques such as group filters [15] to efficiently evaluate the predicates. Once the fragment is determined, the tuple is added to the fragment's corresponding local PSR in A_i.

In the second phase, decomposition, each node will locally apply the decomposition algorithm to F and A_i to produce a smaller matrix, A_i'. The specific decomposition algorithm used is dependent on the type of aggregate function being computed. In Section 3 we present the basic algorithm that applies to all functions. Section 4 shows an algorithm that can be used for linear aggregate functions and in Section 5 we show a family of heuristic algorithms that work for duplicate insensitive functions.

We require that every node in the system use the same F matrix for decomposition. The F matrices must be the same so that every entry in A_i' has the same meaning, or in other words, contains a piece of the answer to same set of queries. Nodes that do not have any tuples for a particular fragment will have an empty PSR in A_i. In Section 8.1, we explain how to synchronize F on all nodes; for duplicate insensitive aggregate functions, we are able to eliminate this requirement altogether.

In the third phase, global aggregation, we aggregate each of the A_i''s over all nodes in the system to produce the global A'. Since we want to maintain the load balanced property of the non-sharing case, we aggregate each entry/fragment in A' separately in its own aggregation tree. Once the final value has been computed for an entry of A' at the root of its respective aggregation tree, the PSR is sent to a single coordinator node for reconstruction.

The fourth phase, reconstruction, begins once the coordinator node has received each of the globally computed A' entries. Using the F matrix (or its decomposition) the answer to all queries can be computed. The reconstruction algorithm is related to the specific decomposition algorithm used, and is also described in the respective sections.

We take a moment to highlight the basic costs and benefits of this method. Both the sharing and no-sharing methods must disseminate every query to all nodes. This cost is same for both methods and is amortized over the life of the continuous query. Our method introduces the cost of having all nodes agree on the same binary F matrix, the cost to collect all of the A' entries on a single node, and finally the cost to disseminate the answer to each node that issued the query. The benefit is derived from executing fewer global aggregations (in the third phase). The degree of benefit is dependent on the data/query workload. In Section 8.1 we analytically show for which range of scenarios this method is beneficial.
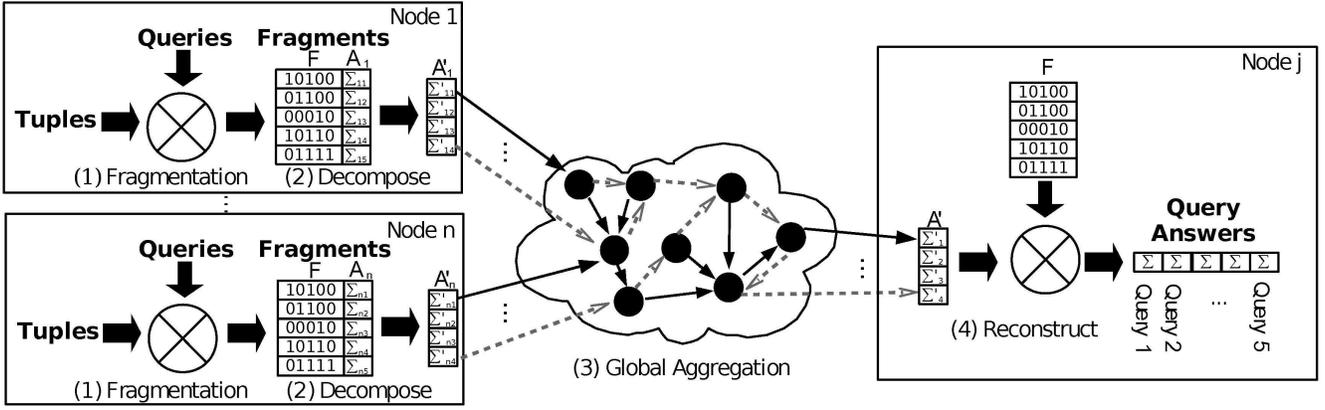
Figure 1: Tuples are first aggregated by fragment (1) into a local A_i PSR. F and A_i are then decomposed (2) to form A_i'. Each entry in A_i' is then aggregated over all nodes (3) in separate aggregate trees. The final global value for each entry in A' sent to some node j. Node j can then reconstruct (4) the answers to every query and distribute the result.

3 General Decomposition Solution

Our first algorithm, basic decomposition, applies to all aggregation functions, and directly follows the optimization we presented the intuition for in the previous section. Our aim is to find the smallest set of basis rows, such that each row is exactly the disjunction of two or more basis rows that are non-overlapping – i.e., their conjunction is empty. If the basis rows were to overlap, then a tuple would be aggregated multiple times for the same query.

Formally, we want to find the basis rows in F under a limited algebra. Standard boolean logic does not allow us to express the requirement that basis rows be non-overlapping. Instead, we can define an algebra using

a 3-valued logic (with values of 0, 1, and I for "invalid") and a single binary operator called ONCE. The output of ONCE is 1 iff exactly one input is 1. If both inputs are 0, the output of ONCE is 0, and if both inputs are 1 the output is I. Using this algebra, the minimal set of rows which can be ONCEed to form every row in F is the minimal basis set, and our target solution. The I value is used to prevent any tuple from being counted two or more times for the same query.

The exhaustive search solution is prohibitively expensive, since if each row is q bits there are 2^{2^q} possible solutions. While this search space can be aggressively pruned, it is still too large. Even a greedy approximation is very expensive computationally, since there is a total of 2^q choices (the number of possible rows) at each step – simply enumerating this list to find the locally optimal choice is clearly impractical.

To approach this problem, we introduce a simple heuristic that attempts to find basis rows using the existing rows in F. Given two rows, i and j, if j is a subset of i then j is covering those bits in i that they have in common. We can therefore decompose i to remove those bits that are in common. When we do that, we need to alter A by adding the PSR from i's entry to j's entry.

We can define a DECOMPOSE operation as:

DECOMPOSE(F, A_i, i, j):
if $(i \neq j)$ AND $(\neg F[i] \& F[j] = 0)$ **then**
  $F[i] = F[i] \text{XOR} F[j]$
  $A_i[j] = A[j] + A[i]$
else return invalid

The DECOMPOSE operation can be applied to pairs of rows until no progress can be made. The following algorithm, will transform F and A_i into F' and A_i':

BASIC DECOMPOSITION(F, A_i):
boolean progress = true
while progress = true
  progress = false
  for all rows $i \in F$
    for all rows $j \in F$
      if Decompose(F, A_i, i, j) $\neq$ invalid
        then progress = true
  for all rows $k \in A_i$
    if $|F[k]| = 0$ **then**
      $A_i[k] = \emptyset$ \\ *rows in F with all 0's*

Reconstruction is straightforward since $A_i'^\mathsf{T} \times F' = A_i^\mathsf{T} \times F$.

The running time of the basic decomposition algorithm is $O(f^3)$, where f is the number of rows in F. Since the basic decomposition is searching a small portion of the search space, it is not expected to produce the smallest basis set. Furthermore, it is the only algorithm we present that can produce an answer worse than no-sharing. The algorithm starts with f basis rows, where f can be greater than q, and attempts to reduce the size of this initial basis. This reduction may not always be sufficient to find a basis that is smaller than or equal to q (although one such basis must exist). In these cases we revert to a $q \times q$ identity matrix which is equivalent to a no-sharing solution. However, this simple

algorithm does provide a foundation for our other solutions.

4   Linear Aggregate Functions

If the aggregate function is linear, such as COUNT, SUM, or AVERAGE, we are no longer constrained to using the limited algebra from the previous section. Instead, we can treat the matrix entries as real numbers and use linear algebra techniques akin to Gaussian Elimination, adding and subtracting rows in F from each other, and multiplying these rows by scalars. Our goal of reducing the size of A_i can therefore be accomplished by finding the minimal set of linearly independent rows F' in F, or the rank of F. By definition F can be reconstructed from F', so we can create A'_i from A_i at the same time and still correctly answer every query during the reconstruction phase.

For example, suppose we are calculating the COUNT for these five queries with this F and A_i matrix:

$$F = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} A_i = \begin{bmatrix} 13 \\ 54 \\ 24 \\ 78 \\ 32 \end{bmatrix}$$

The answer to the first query (in the leftmost column) is $13 + 54 + 78 + 32$ or 177. The complete solution matrix can be computed using $A_i^{\mathsf{T}} \times F$.

It turns out that we can express F and A_i using only four rows:

$$F' = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} A'_i = \begin{bmatrix} 177 \\ -30 \\ 37 \\ 13 \end{bmatrix}$$

Using F' and A'_i we can still produce the correct solution matrix, using $A_i'^{\mathsf{T}} \times F'$. In this example we used Gaussian Elimination on F to find the smallest set of basis rows. We will now discuss how to solve this problem using more efficient algorithms.

In numerical computing, *rank-revealing factorizations* are used to the find the minimal set of basis rows. We will apply three well-studied factorizations to our problem: the LU, QR, and SVD decompositions. These algorithms will decompose F into two or more matrices that can be used in local decomposition to transform A_i into A'_i and then to reconstruct A' into the query answers at the coordinator node. These factorization methods and their implementations are well studied in the numerical computing literature [1]. We now present formulations for utilizing these factoring methods.

An LU algorithm factors F into a lower triangular matrix L and an upper triangular matrix U such that $L \times U = F$. In the decomposition phase we can form A'_i using $A_i^{\mathsf{T}} \times L$ and remove any entries in A'_i whose corresponding row in U is

composed of all zeros. Reconstruction at the coordinator is simply $A' \times U$. We can safely remove the entries in A'_i whose corresponding row in L is all zeros because in reconstruction those entries will always be multiplied be zero and thus do not contribute to any results. During reconstruction we insert null entries in A' as placeholders to insure the size of A' is correct for the matrix multiplication.

Using QR factoring is very similar to using LU. In this case, the QR algorithm factors F into a general matrix Q and an upper triangular matrix R such that $Q \times R = F$. We form A'_i using $A_i^{\mathsf{T}} \times Q$ and remove any entries in A'_i whose corresponding row in R is composed of all zeros. Reconstruction is accomplished using $A' \times R$.

SVD factors F into three matrices, U, S, and V^{T}. A'_i is formed in decomposition using $A_i^{\mathsf{T}} \times U \times S$. Using this method, we remove entries from A'_i whose corresponding row in S is zero. Reconstruction is accomplished by computing the product of A' and V^{T}. With all three algorithms, the factorization of F is deterministic and therefore the same on all nodes, allowing us to aggregate A'_is from all nodes before performing reconstruction.

These algorithms all have a running time of $O(m * n^2)$ where m is the size of the smaller dimension of F and n is the larger dimension. In addition, all three methods would be optimal (finding the smallest basis set and thus reducing F and A_i to the smallest possible sizes) using infinite precision floating point math. However, in practice these are computed on finite-precision computers which commonly use 64 bits to represent a floating point number. Factorization requires performing many floating point multiplications and divisions which may create rounding errors that are further exacerbated through additional operations. While LU factorization is especially prone to the finite precision problem, QR factoring is less so, and SVD is the least likely to produce sub-optimal reductions in A''s size. Due to this practical limitation, the factorization may not reach the optimal size. In no case will any of these algorithms produce an answer that requires more global aggregations than the no-sharing scenario. In addition, these rounding error may introduce errors in A' and therefore perturb the query results. However, these algorithms, in particular SVD, is considered robust and used in many applications.

5   Duplicate Insensitive Aggregate Functions

The previous algorithms preserve the invariant that each tuple that satisfies a particular query will be aggregated exactly once for that query. However, some aggregate functions, such as MIN and MAX, will still produce the same answer even if a tuple is aggregated more than once. We can take advantage of this property when decomposing F and achieve a higher communication savings compared to the previous

algorithms. Consider this simple example:

$$F = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} \quad F' = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

We notice that the fifth row of F is equal to the OR of the second and third (or second and fourth, or third and fourth). Thus we can define a matrix F' that removes this redundant row. The corresponding operation to the A matrix is to aggregate the fifth entry with the second entry, aggregate the fifth entry with the third entry, and then remove the fifth entry. Intuitively, this is moving the data from the fifth fragment to both the second and third fragments.

Similar to the previous sections, the goal is to find the minimum number of *independent rows*. But in this case, the independent rows are selected such that all rows in F can be obtained by combining rows in the basis using only the standard OR operation, rather than ONCE.

This problem is also known as the *set basis* or *boolean basis* problem. The problem can be described succinctly as follows. Given a collection of sets $S = \{S_1, S_2, ...S_s\}$, a basis B is defined as a collection of sets such that for each $S_i \in S$ there exists a subset of B whose union equals S_i; the set basis problem is to find the smallest such basis set. Our problem is the same, where S = rows of F and B = rows of F'. The set of possible basis sets is 2^{2^n} where n is the number of elements in $\bigcup S$. This problem was proved NP-Hard by Stockmeyer [17], and was later shown to be inapproximable to within any constant factor [13]. To our knowledge, ours is the first heuristic approximation algorithm for the general problem. In [12] Lubiw shows that the problem can be solved for some limited classes of F matrices, but these do not apply in our domain.

As with the general decomposition problem in Section 3, the search space of our set basis problem is severely exponential in q. To avoid exhaustive enumeration, our approach for finding the minimal basis set, F', is to start with the simplest basis set, a $q \times q$ identity matrix (which is equivalent to executing each query independently), and apply transformations. The most intuitive transformation is to OR two existing rows in F', i and j, to create a third row k. Using this transformation (and the ability to remove rows from F' one could exhaustively search for the minimal basis set. This approach is obviously not feasible.

We apply two constraints to the exhaustive method in order to make our approach feasible. First, after applying the OR transformation, at least one of the existing rows, i or j, is always immediately removed. This ensures that the size of the basis set never increases. Second, we maintain the invariant that after each transformation the set is still a valid basis of F.

We can now formally define two operations, BLEND and COLLAPSE which satisfy these invariants. Given a matrix F

and a basis F' for F, both operations overwrite a row $F'[i]$ with the OR of row $F'[i]$ and another row $F'[j]$. COLLAPSE then removes row j from F', whereas BLEND leaves row j intact. After performing one of these operations, if the new F' still forms a basis for F then the operation is valid; otherwise the original F' is kept.

COLLAPSE is the operation that achieves a benefit, by reducing the size of the basis set by one. COLLAPSE is exploiting the co-occurrence of a bit pattern in F. However, it may not be valid to apply COLLAPSE until one or more BLEND operations are performed. The intuition for this is that when the bit pattern in some original row can be used in multiple basis rows, BLEND preserves the original row so that it can be used as, or part of, another basis row. Consider matrix F, and the following *invalid* COLLAPSE transformation:

$$F = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad F' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

We cannot directly COLLAPSE rows one and four in F' as shown above. The resulting F' is no longer able to reconstruct the first or fourth rows in F via any combination of ORs; we call such a transformation "invalid". However, if we first BLEND rows two and four (leaving row four), we can then COLLAPSE rows one and four, as shown next:

$$F' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Using these two operations we can search a subset of the overall search space for the minimal basis set. A simple search algorithm, called BASIC COMPOSITION, performs BLEND or COLLAPSE in random order until no more operations can be executed. The pseudo-code is shown below:

```
BASIC COMPOSITION(F)
F' = qxq identity matrix
boolean progress = true
while progress = true
  progress = false
  for all rows i ∈ F'
    for all rows j ∈ F'
      if i ≠ j then
        if COLLAPSE (F, F', i, j) ≠invalid then
          progress = true
          break to while loop
        if BLEND (F, F', i, j) ≠invalid then
          progress = true
          break to while loop
```

A'_i can be calculated by aggregating together each element in A_i that corresponds to a row in F which is equal to or a superset of the A'_i entry's corresponding F' row.

There are three key limitations of this algorithm:

- Once an operation is performed it can not be undone: both operations are non-invertible and there is no backtracking. This limits the overall effectiveness of finding the minimal basis set since the algorithm can get trapped in local minima.

- The random order in which operations are performed can determine the quality of the local minimum found.

- At any given point there are $O(f^2)$ possible row combinations to choose from. Finding a valid COLLAPSE or BLEND is time consuming.

In effect, the algorithm takes a single random walk through the limited search space. For some workloads, the optimal solution may not even be attainable with this method. However, while this heuristic algorithm gives no guarantees on how small the basis set will be, it will never be worse than the no-sharing solution. We will show in Section 7 that this heuristic is often able to find 50% of the achievable reductions in the size of the basis set, but its running time is extremely long.

Refinements. Our first refinement takes a slightly different approach. Instead of optimizing every query at once, we incrementally add one query at time optimizing as we go. The two key observations are (1) that a valid covering for $q-1$ queries can cover q queries with the addition of a single row which only satisfies the new query and (2) the optimal solution for q queries given an optimal basis solution for $q-1$ queries and the single basis row for the qth query will only have up to one valid COLLAPSE operation.

Using these observations we can define the ADD COMPOSITION algorithm which incrementally optimizes queries one at a time:

ADD COMPOSITION(F, F', $start$)
Require: F' has $start$ columns
  let q = the number of queries \\ *columns in F*
  let f = the number of rows \\ *rows in F'*
  for $c = start + 1$ **up to** q
    Expand F' to $(f + 1) \times c$ with 0's
    $F'[f + 1][c] = 1$
    F_c = Project(F,c) \\ *See Following Algorithm*
    Optimize(F_c,F',$f + 1$)
  return F'

PROJECT(S, $columns$)
  for all rows $i \in S$
    for all cols $j \in S$
      if $j \leq columns$ **then**
        $S''[i][j] = S[i][j]$
      else $S''[i][j] = 0$
    S' = unique rows in S''
  return S'

The OPTIMIZE step in ADD COMPOSITION is very similar to the repeat loop in BASIC COMPOSITION. It has a *search loop* that continues looking for combinations of rows that can be used in a COLLAPSE or BLEND operation until there are no such combinations. OPTIMIZE has two key improvements over the BASIC COMPOSITION. First, COLLAPSES and BLENDS are not considered if they combine two old (optimized) rows. Second, since only one row was added to F', once a COLLAPSE is performed the optimization is over and the search loop is stopped, since no additional COLLAPSES will be found. As shown in Section 7 this method is considerably faster and still equally effective at finding a small basis set compared to the Basic Composition algorithm.

We consider the following three dimensions for search loop strategies:
- **Number of operations per iteration:**
  - *O:* Perform only one operation per search loop and then restart the loop from beginning.
  - *M:* Perform multiple operations per search loop only restarting after every combination of rows are tried.

- **Operation preference:**
  - *A:* Attempt COLLAPSE first, but if not valid attempt BLEND before continuing the search.
  - *R:* Perform all COLLAPSES while searching, but delay any BLENDS found till the end of the search loop.
  - *S:* First search and perform only COLLAPSE operations, then search for and perform any BLENDS. This requires two passes over all row pairs per loop.

- **Operation timing:**
  - *W:* Execute operations immediately and consider the new row formed in the same loop.
  - *D:* Execute operations immediately but delay considering the new row for additional operations till the next loop.
  - *P:* Enqueue operations till the end of the search loop and then execute all operations.

The BASIC COMPOSITION algorithm shown uses the O/A strategy. The algorithm performs one operation per iteration of the outer loop. So after each operation, it will begin the search again from the beginning. The algorithm favors COLLAPSE by attempting that operation first. The operator timing dimension is not relevant for strategies that only perform one operation per iteration. Note that the BASIC COMPOSITION can be modified to use any of the possible search strategies. In the evaluation section we only show the strategy that performed the best in our experiments, M/A/W.

There are only twelve search strategies possible using the three dimensions evaluated (when performing only one operation per search loop, operation timing is not relevant). All twelve are experimentally evaluated in Section 7.

6 Potential Gains

Before we evaluate the effectiveness of our techniques experimentally, we explore the analytical question of identifying query/data workloads that should lead to significant beneficial sharing, and quantifying that potential benefit. This will provide us a framework for evaluating how close our "optimization" techniques come to a true optimum. In this section, we show that there are cases where sharing leads to arbitrarily high gain, given a sufficient number of queries. We present two constructions, one designed for duplicate insensitive query workloads, the other for duplicate sensitive workloads. Our goal is to construct a workload that maximizes the sharing or benefit potential. We define the total gain, G_t as:

$$G_t = 1 - (\text{\# aggregates executed} \div \text{\# queries answered})$$

We also define the fragment gain which is the gain over computing each fragment, s as:

$$G_f = 1 - (\text{\# aggregates executed} \div \text{\# fragments})$$

The total gain, G_t, is the most important metric, since an effective decomposition algorithm can translate this sharing potential into a proportional amount of network bandwidth savings. The fragment gain, G_f is the benefit over computing every fragment in F.

6.1 Duplicate Insensitive

To maximize the sharing potential we start with b base queries $(b_1, b_2, b_3, ...b_b)$ and data that satisfies every conjunctive combination of the b queries $(\{b_1\}, \{b_2\}, \{b_3\}, ...\{b_1, b_2\}, \{b_1, b_3\}, ...\{b_1, b_2, b_3, ...b_b\})$ such that we have $2^b - 1$ fragments (the -1 is for data that satisfies no queries). At this stage, no sharing is beneficial since only b aggregates are actually needed (one for each query).

Using the initial b queries, we can write an additional $2^b - 1 - b$ queries by combining them via *disjunction*, i.e. query x matches data that satisfies query b_1 and b_2, query y matches data satisfying b_2 or b_3, etc. One such additional query is outlined in Figure 2(a). In this case there are 2^b such combinations from which we subtract the original b queries and the combination that is the disjunction of the empty set of queries. The additional queries do not introduce any additional fragments.

These new $2^b - 1 - b$ queries can be answered if we have the answers to the original b queries. Since the aggregate functions we consider here are duplicate insensitive, the disjunction of multiple queries is simply their aggregation. So if we compute the aggregates for the original b queries, we can clearly answer the original b queries plus the new $2^b - b - 1$ queries for a total of $2^b - 1$ queries. Thus, $G_t = G_f = 1 - \frac{b}{2^b - 1}$. As $b \to \infty$ the gain approaches 1 which is maximal.
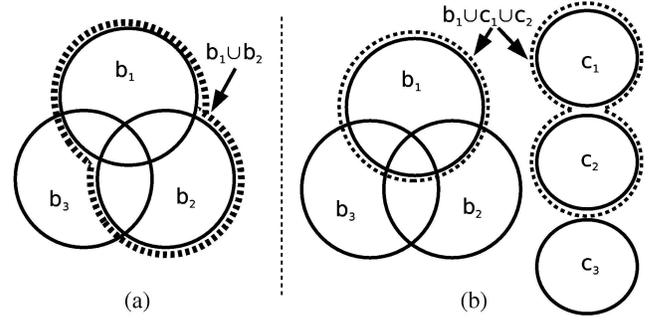

Figure 2: Example Venn diagrams for duplicate insensitive construction (a) and the duplicate sensitive construction (b). In (a) the additional query $b_1 \cup b_2$ is outlined. In (b) the additional query $b_1 \cup c_1 \cup c_2$ is outlined.

The intuition behind this construction is that queries that are the disjunction of other queries lead to sharing opportunities. While the first b base queries have significant amount of overlap, the overlap is not precise creating additional fragments. It should be noted that none of the $2^b - 1$ fragments created from the b base queries are actually used to answer any queries, instead the b base queries are computed directly and used to compute the additional $2^b - 1 - b$ queries. This is only possible because the aggregation functions are duplicate insensitive and the overlap in data between the b base queries does not affect the answer for the additional queries.

Furthermore, it is not necessary that the b base queries are explicitly requested by users. If only the additional queries were issued, those queries could still be answered using just b global fragments. This means that the gain is realized when the query set is just the disjunction of a smaller number of overlapping fragments.

6.2 Duplicate Sensitive

This construction is similar to the previous construction, with b base queries and $2^b - 1$ fragments. Now we add c non-overlapping queries such that data that satisfies one of the c queries and does not match any other query (from b or c). Thus, there are c additional fragments for a total of $b + c$ fragments.

We now add $2^c - 1 - c$ additional queries based solely on the c non-overlapping queries by taking the disjunction of every possible combination of the c queries. These queries can be answered by aggregating the answers from the original c queries. Note, this does not count any tuple twice since the c queries were non-overlapping.

We also add $(2^c - 1) \times (b)$ more queries by taking the disjunction of every possible combination of the c queries and exactly one query from the b base queries. For example, we take $c_1 \cup b_1$, $c_2 \cup b_2$, $c_1 \cup c_2 \cup b_1$ and $c_1 \cup c_2 \cup b_2$, etc. One such additional query is outlined in Figure 2(b). Since each of these additional queries is only the disjunction of one query from b, there is still no overlap, so no data is counted

multiple times.

In summary we have $b + c + 2^c - 1 - c + (2^c - 1) \times b$ queries which could be answered using $b+c$ fragments. This leads to a total gain of $1 - \frac{b+c}{2^c - 1 + b \times 2^c}$, and fragment gain of $1 - \frac{b+c}{2^b - 1 + c}$. As b and c approach infinity, the total and fragment gains approach 1 which is maximal.

Intuitively, the c queries are the source of sharing, since we are able to construct many additional queries that are the disjunction of multiple base c queries. The b queries are the source of the fragment gain, since the overlap they create increases the number of fragments that are not needed.

7 Experimental Evaluation

In this section we evaluate the performance of the various decomposition methods we have presented. Rather than focus on a specific workload from a speculative prototype system, we pursue an experimental methodology that allows us to map out a range of possible workloads, and we evaluate our techniques across that range.

We present a random workload generator based on our analysis of the gain potential in the previous section. This generator allows us to methodically vary the key parameters of interest in evaluating our techniques: the workload size, and the degree of *potential* benefit that our techniques can achieve. Within various settings of these parameters, we then compare the relative costs and benefits of our different techniques. After describing our workload generator, we present our experimental setup and our results.

7.1 Workload Generators

We designed a workload generator that allows us to input the *desired size* and *total gain* for a test F matrix. By controlling the total gain we are able to test the effectiveness of our algorithms. Using the combination of the two knobs we can explore various workloads. We have two generators, one for duplicate sensitive aggregates and one for duplicate insensitive aggregates, that create test F matrices. The constructions from the previous section are used to develop these generators.

For the duplicate insensitive generator we can calculate the number of basis rows, b, the number of fragments, f, and the number of queries, q based on the desired matrix size and gain. Each of the b basis rows maps to one of the b base queries in the constructor. Instead of generating all $2^b - 1$ fragments, we uniformly at randomly select the f fragments from the set of possible fragments. Analogously, we uniformly at randomly select unique additional columns (queries) from the set of up to $2^b - b - 1$ possible additional queries. The generation is finalized by randomly permuting the order of the rows and columns.

This construction gives us a guarantee on the upper bound for the minimum number of basis rows needed, b. The optimal answer may in fact be smaller if the rows selected from the set of $2^b - 1$ can be further reduced. Since the rows are chosen randomly, such a reduction is unlikely. In our experiments, we attempt to check for any reduction using the most effective algorithms we have.

The duplicate sensitive generator works much the same, except with the addition of the c basis rows. The additional columns (queries) are generated by ORing a random combination of the c base queries and up to one of the b base queries. Values for the number of b and c queries are randomly chosen such that their sum is the desired number of basis rows and such that b is large enough to ensure enough bitmaps can be generated and c is large enough that enough combination of queries can be generated.

Also note that the original b (and c) queries remain in the test matrix for both generators; while this may introduce a bias in the test, we are unable to remove these queries and still provide a reasonable bound on the optimal answer. Without knowing the optimal answer it is hard to judge the effectiveness of our algorithms.

7.2 Experimental Setup

We have implemented in Java all of the decomposition algorithms presented in the previous sections. Our experiments were run on dual 3.06GHz Pentium 4 Xeon (533Mhz FSB) machines with 2GB of RAM using the Sun Java JVM 1.5.06 on Linux. While our code makes no specific attempt to utilize the dual CPUs, the JVM may run the garbage collector and other maintenance tasks on the second CPU. All new JVM instances are first primed with a small matrix prior to any timing to allow the JVM to load and compile the class files.

For the LU/QR/SVD decompositions we utilize the JLAPACK library, which is an automatic translation of the highly optimized Fortran 77 LAPACK 2.0 library. We also tested calling the Fortran library from C code. Our results showed that the Java version was about the same speed for the SVD routines (in fact slightly faster in some instances) while the more optimized LU and QR routines were about twice as slow on Java. Overall, the runtime differences are minor and do not effect our conclusions on relative speed or effectiveness so we only present the results from the Java version.

We have three metrics:

- the *relative effectiveness* (which is equivalent to the relative decrease in network bandwidth used for computing the aggregates)

- the *running times* of the decomposition routine

- the *absolute size* of the resulting matrix A' which is directly proportional to the network bandwidth

In particular, the relative effectiveness is based on the resulting size of A', the estimated optimal answer k and the number of queries q. It is defined as $(q - |A'|) \div (q - k)$ or the ratio of attained improvement to that of an optimal algorithm.

We vary the ratio of the number of fragments to queries (whether the test matrix is short, square, or long) from $1/2$ to 2. We repeat each experiment ten times with different test matrices of the same characteristics (size and total gain); the results presented include the average and plus/minus one standard deviation using error-bars.

7.3 Results

We first present the result from the linear decomposition algorithms, which prove effective and fast for linear aggregates. Next, we show results for the duplicate insensitive heuristics where we highlight the best and worst performing variants. Finally we discuss the results for the basic decomposition algorithm.

Linear Aggregates. Our first set of experiments evaluate the linear aggregate decompositions using the duplicate sensitive workload generator. In Section 4 we noted that LU, QR, and SVD can be used to compute to A'. They differ in their running times, and in practice there is a concern that numerical instability (via rounding during repeated floating-point arithmetic) can cause the techniques to incorrectly solve for the basis, and produce an inefficient F'. Figure 3 shows the resulting size of A', the overall effectiveness, and the running time for the three algorithms using square matrices (500 queries with 500 fragments).

QR and SVD give always optimal results by finding the lowest rank and therefore the smallest A'. LU lagged in effectiveness due to sensitivity to precision limitations, with low effectiveness for matrices that had small potential gain and near optimal effectiveness for matrices that had high potential. As expected, LU and QR are substantially faster than SVD in our measurements by about an order of magnitude. Figure 4 shows that runtime increases polynomially ($O(q^3)$) as the size of the test matrix is increased.

The trends remain the same when the shape of the test matrix is changed. These results are summarized in the Appendix. QR and SVD remain optimal and LU has an overall effectiveness ranging from 50-85%. As expected, the running times increase for matrices with additional rows and decrease for matrices with fewer rows.

In summary, QR achieves the best tradeoff of effectiveness and speed. While SVD has been designed to be more robust to floating point precision limits, QR was able to perform just as well on this type of binary matrix. LU has no benefit, since it is just as fast as QR, but not nearly as effective in finding the lowest rank.

Duplicate Insensitive Aggregates. Our second set of tests evaluate the composition family of heuristics using the duplicate insensitive workload generator. In Figure 5 we show the results. For clarity, we include a representative selection of the algorithms, including the BASIC COMPOSITION and the ADD COMPOSITION using five strategies (OR, MAP, MAD, MAW, and MRW). The strategies for ADD COMPOSITION were chosen since they include: (1) the best strat-
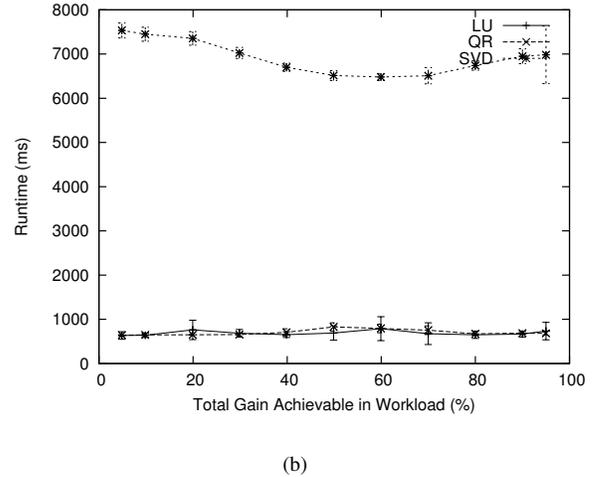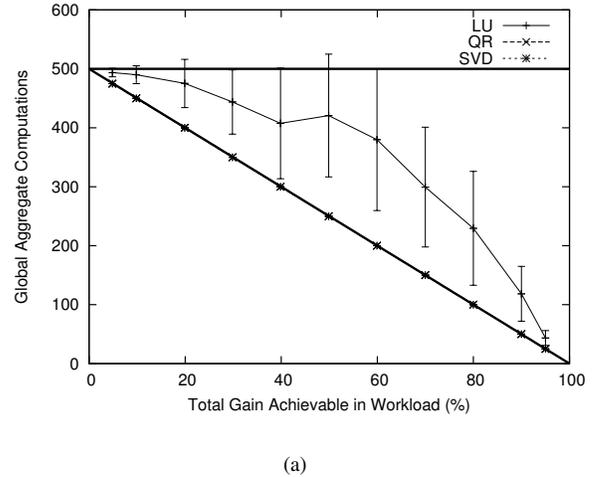

(a)

(b)

Figure 3: For 500x500 test matrices: (a) shows the resulting size of A'. The solid line at y=500 represents the starting size and the lower solid line represents optimal. QR and SVD are always optimal and are drawn on top of the lower line. (b) shows the running time for each.
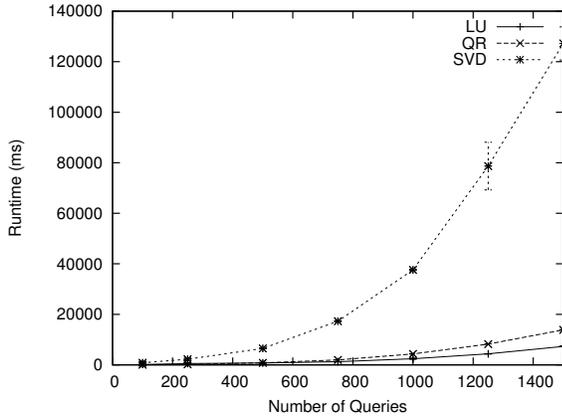
Figure 4: The running time as the size of square matrices are increased.

egy when used with BASIC COMPOSITION, (2) the worse performing, (3) the best performing and (4) two strategies similar to the best performing. The strategies not shown have similar shaped curves falling somewhere in the spectrum outlined by those shown. All strategies are defined in Section 5. The results for all algorithms are summarized in the Appendix.

The results show that ADD COMPOSITION with the MAP search strategy is both the most effective and the fastest, although not much more effective than with the OR strategy (which is substantially slower). This is somewhat surprising given how different the MAP and OR strategies seem. Also note that in most cases the relative effectiveness and the running time are inversely correlated. This indicates that some algorithms spend a lot of time searching and producing little benefit.

As explained in Section 5 the gain is revealed through the COLLAPSE operation. However, before COLLAPSE can be performed often a number of BLEND operations are needed before a COLLAPSE can be used. Search strategies that search for both COLLAPSE and BLEND at the same time tend to do better than strategies that search for more and more BLENDs after each other.

For example the MSW search strategy will first search for any possible COLLAPSE operations, and then search for BLEND operations separately. As an operation is performed the resulting row is considered for further operations in the same search loop. Even though COLLAPSEs are performed before BLENDs, once the strategy begins performing BLENDs it will continue to exclusively perform them until no more can be found. As a result, it gets stuck in this phase of the search loop. Even worse, it performs so many BLEND operations that they block future COLLAPSE operations and find a poor local minimum. This strategy often finds a local minimum and ends after it executes only two or three search loops.
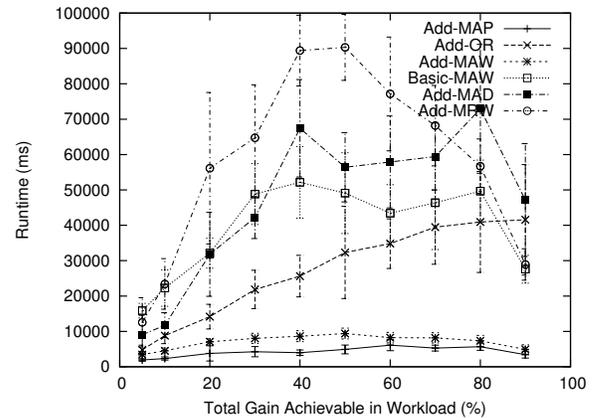
In contrast, the OR and MAP strategies are quick to search
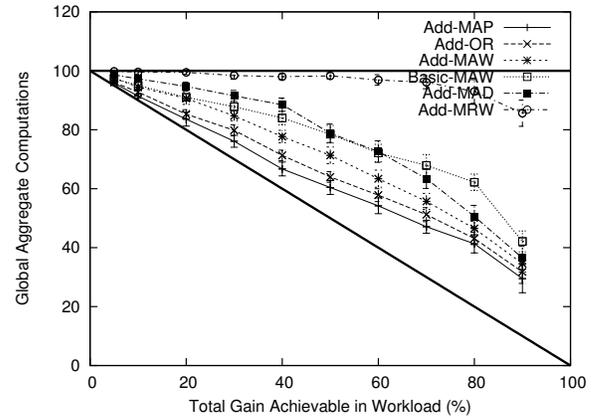
(a)
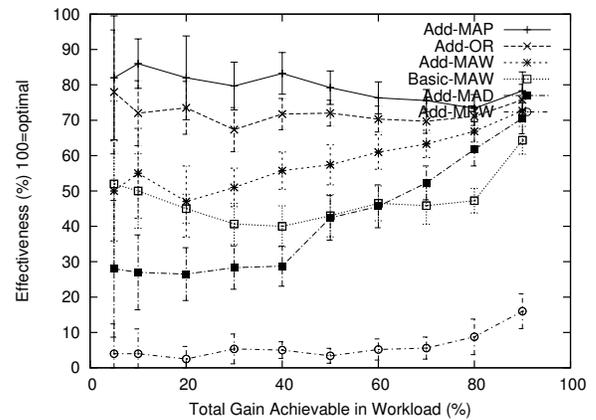
(b)

(c)

Figure 5: For 100x100 test matrices (a) shows the resulting size of A'. The solid line at y=100 represents the starting size and the lower solid line represents optimal. (b) shows the relative effectiveness. (c) shows the running time for each.
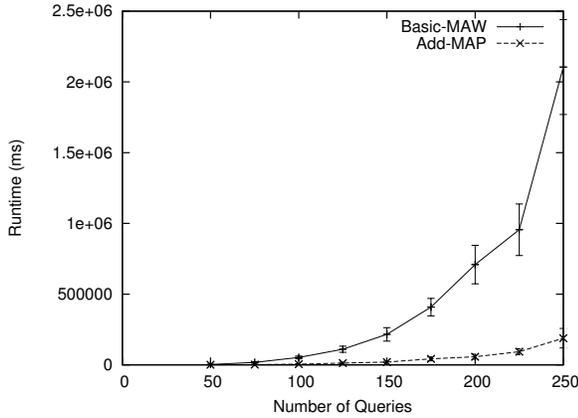
Figure 6: The running time as the size of a square matrix is increased.

for more COLLAPSE operations after performing any operation. In the case of MAP, all possible operations with the given set of rows is computed, and they are then executed without further searching. While this tends to need many search loops, the strategy will not get caught in a long stretch of BLENDs. In the case of OR, after every operation the search loop ends, and the search restarts. This strategy prevents getting stuck in the BLEND phase, but also wastes time continually searching the same portion of the search space over and over again after each operation. This causes the OR strategy to be considerably slower than the MAP strategy.

Figure 6 shows the running times of the fastest basic composition and additive composition strategies, for various sized square matrices. Unfortunately, none of the algorithms scale well as the matrix size is increased. However, the additive composition scales considerably better than the basic composition algorithm. Effectiveness, not shown, remains the same or slightly increases as the size of the matrix increases.

Note that the super-linear scaling is not unexpected. The problem of finding a minimal boolean basis has been shown to be equivalent to finding the minimal number of maximal cliques in a bipartite graph [12]. Finding the maximal biclique is not only NP-Hard, but also is known to be hard to approximate. Our solution, while not fully exhaustive, considers a very large number of possibilities and produces near-optimal answers in our experiments.

In summary, the Add Composition algorithm with the MAP search strategy is the clearly winner. It is 70-90% effective in finding the smallest basis set, and is often the fastest algorithm for duplicate insensitive aggregates.

Basic Decomposition. As expected, our basic decomposition presented in Section 3 which works for all aggregate functions, is ineffective in most situations. Due to space limitations we do not show any results and only summarize our findings. For duplicate sensitive tests, the algorithm can of-

ten produce answers that are worse than no-sharing, generating an A' that has more entries than the number of queries, and in rare cases showing modest sharing of 5-20% optimal. The algorithm performs best in cases where there is large sharing potential. Perhaps the only redeeming characteristic of the algorithm is that it is fast, running faster than a half second for 500 queries, and only a few seconds for 1500 queries. For duplicate insensitive tests, the algorithm finds sharing potential extremely rarely, but runtime remains the same. This is expected, since the algorithm makes no attempt to exploit the duplicate insensitive property.

Given these results, it is clear that this general-purpose technique should not be used when our special-purpose solutions can be used (i.e., for duplicate insensitive and linear aggregates.) Non-linear, duplicate-sensitive aggregates appear to be an extremely difficult family to optimize in our context.

8 Practical Matters

In this section we discuss how we ensure that every node has the correct F matrix, whether through explicit or implicit communication, and its associated network overhead. We then discuss how to extend our methods to work for a larger class of complex queries.

8.1 Synchronizing F Across the Network

In order to ensure the PSRs in A' that are communicated from one node to another are correctly decoded we must guarantee that every node has the same F matrix. Otherwise, during the global aggregation or reconstruction phases, the PSRs in A' may be incorrectly aggregated causing the query results to be wrong. This is very important for correctness of some decomposition algorithms such as the linear algebra routines LU, QR, and SVD. For the other decomposition algorithms presented there is an optimization to the architecture that eliminates this requirement. We first describe a simple method for ensuring all nodes have the same F and then describe the optimization.

At the end of every aggregation window (after the node has collected all the raw data necessary to compute the aggregates for that window) each node, i, computes its local F matrix, F_i. Since each node may have a different distribution of data, the matrix F_i at node i may differ from matrix F_j at node $j \neq i$. The global F is the set union of the rows in all local F_i's.

This can be computed like any other aggregate using a tree. All the leaves of the tree send their complete F_i to their parents. Their parents compute the union over all their children, and send the result to their parent. At the root of this aggregation tree, the global F is computed. The global F is then multicast to every node on the reverse path of the aggregation tree.

For subsequent windows only additions to F need to be transmitted up or down the aggregation tree. Deletions can also be propagated up the aggregation tree, however any node along the path can stop the propagation (which prevents a change to the global F) if it has at least one other child (or itself) still needing that row. The addition or deletion of a query will also change F. Query (column) deletions require no communication (every node simply removes the column for F). The addition of a query (column) affects every row in F, but in a limited fashion. Each row is either extended with a 1, a 0, or both (which requires duplicating the old row). This can be compactly transmitted as a modification bitmap with two bits per existing row. The global modification bitmap is the OR of every node's individual modification bitmap which can also be efficiently computed as an aggregate.

Once all nodes have the global F, the general computation of the query aggregates can begin. This synchronization method has the negative effect of delaying all results for at least the duration of one global aggregation plus one global multicast. In practice, the actual delay must be sufficiently long to accommodate worst case delays in the network.

The exact communication cost of this method is dependent on the query/data workload. However, given a constant set of q queries and a set of n nodes, we can show the worst case cost of synchronizing F for each additional bitmap, and for how many windows the system must remain unchanged to recoup the cost.

The worst case communication cost occurs if at least every leaf node in the aggregation tree requires the addition of the same new row in a given aggregation window. In this situation every node will need to transmit the new row in F up and down the aggregation tree which yields a cost of $2 * n * q$ bits per row. If only one node requires the new row the cost is roughly $n * q + \log(n) * q$ as only one node is sending data up the aggregation tree.

Assume the size of each PSR is p bits. The savings realized from sharing will never be less than the eventual total gain, G_t. During each window, $(1 - G_t) * q$ aggregates are being computed instead of q queries in the no-sharing scenario, for a benefit of $((q - (1 - G_t) * q) * p) * n = G_t * q * n * p$ bits per window. We reach the break-even point after $\frac{2 * n * q}{G_t * q * n * p} = \frac{2}{G_t * p}$ windows. If multiple rows must be added at the same time, the number of windows till the break-even point increases proportionally.

The basic decomposition and the algorithms for duplicate insensitive aggregates do not require a global F and can avoid the associated costs. Instead, it is sufficient to annotate every entry in A' with its corresponding binary row in F'. Since every aggregation tree is required to have an identifier (such as a query identifier) to distinguish one tree from another, the basis row entry can be used as the identifier. This is possible since the reconstruction phase does not any require additional information about the decomposition.

While this optimization does not apply to linear aggregates there are other techniques that could be considered. For some query workloads a static analysis of the query predicates may be sufficient to compute a superset of F. This can be further extended to handle the actual data distribution by having nodes compactly communicate which portions of the data space they have. We leave a complete analysis of this optimization for future work.

8.2 Complex Queries

Our query workload to this point might seem limited: sets of continuous queries that are identical except for their selection predicates. In this section we observe that our techniques can be applied to richer mixes of continuous queries, as a complement to other multiquery optimization approaches.

For example, [11, 10] discuss optimizing sharing with queries that have different window parameters. Their methods partition the stream into smaller windows that can later be combined to answer each of the queries. One can view the window-share optimization as query rewriting, producing a set of queries with the same window parameters, which are post-processed to properly answer each specific query. In that scenario, our technique is applied to the rewritten queries. Similarly, queries with different grouping attributes can also be optimized for sharing. In that case, the smallest groups being calculated would be treated as separate partitions of the data that are then optimized separately by our techniques. After processing the results can be rolled-up according to each query's specification.

Our approach does not depend on a uniform aggregation expression across queries. Queries that include multiple aggregate functions, or the same function over different attributes, or queries that require different aggregate functions can be optimized as one in our approach – as long as the same decomposition can be used for all the aggregate expressions. In these cases, the PSR contained in A or A' is the concatenation of each PSR needed to answer all aggregate functions. In those cases where different decompositions must be used (e.g., one function is a MAX and another is a COUNT) then they can be separately optimized and executed using our techniques.

Our results showed that there is a clear choice of which optimization technique to use for most classes of aggregate functions. However, if a function is both linear and duplicate-insensitive, it is unclear which technique to apply. While few functions fall in this category (see Section 2.2), for those functions the selection of algorithm will be dependent on the specific workload. Characterizing the trade-offs among workloads for these unusual functions remains an open problem.

14

9 Conclusion

We have introduced the problem of optimizing sharing for distributed aggregation queries with different selection predicates. We have demonstrated that such sharing can be revealed through the dynamic analysis of a binary fragment matrix capturing the connections between data and query predicates and the algebraic properties of the underlying aggregate functions. For the case of linear aggregates, we show that the sharing potential can be optimally recovered using standard linear-algebra techniques. Unfortunately, for duplicate-insensitive aggregates our sharing problem is NP-hard; thus, we propose a novel family of heuristic search algorithms that is shown to perform well for moderately-sized matrices.

References

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Philadalphia: Society for Industrial and Applied Mathematics., 1994.

[2] S. Cohen and Y. Matias. Spectral bloom filters. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 241–252, New York, NY, USA, 2003. ACM Press.

[3] G. Cormode and M. Garofalakis. "Sketching Streams Through the Net: Distributed Approximate Query Tracking". In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, Trondheim, Norway, Sept. 2005.

[4] G. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi. "Holistic Aggregates in a Networked World: Distributed Tracking of Approximate Quantiles". In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Baltimore, Maryland, June 2005.

[5] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. "Gigascope: A Stream Database for Network Applications". In *Proc. of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 2003.

[6] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing set expressions over continuous update streams. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 265–276, New York, NY, USA, 2003. ACM Press.

[7] M. Garey and D. Johnson. *"Computers and Intractability: A Guide to the Theory of NP-Completeness"*. W.H. Freeman, 1979.

[8] M. B. Greenwald and S. Khanna. "Power-Conserving Computation of Order-Statistics over Sensor Networks". In *Proc. of the ACM Symposium on Principles of Database Systems (PODS)*, Paris, France, June 2004.

[9] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of pier: an internet-scale query processor. In *CIDR*, Jan 2005.

[10] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, June 2006.

[11] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, 2005.

[12] A. Lubiw. The boolean basis problem and how to cover some polygons by rectangles. *SIAM Journal on Discrete Mathematics*, 3(1):98–115, 1990.

[13] C. Lund and N. Yannakakis. "On the Hardness of Approximating Minimization Problems". *Journal of the ACM*, 41(5), Sept. 1994.

[14] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, Dec. 2002.

[15] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Madison, June 2002.

[16] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. "The Design of an Acquisitional Query Processor for Sensor Networks". In *Proc. of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 2003.

[17] L. J. Stockmeyer. The minimal set basis problem in NP-complete. Technical Report RC 5431, IBM Research, May 1975.

[18] N. Trigoni, Y. Yao, A. J. Demers, J. Gehrke, and R. Rajaraman. Multi-query optimization for sensor networks. In *Distributed Computing in Sensor Systems*

(DCOSS), pages 307–321, Marina del Rey, CA, 2005. Springer.

[19] P. Yalagandula and M. Dahlin. SDIMS: A Scalable Distributed Information Management System. In *Proc. of the ACM SIGCOMM Conference*, Portland, Oregon, 2004.

[20] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 299–310, New York, NY, USA, 2005. ACM Press.

APPENDIX

Data for all algorithms averaged over all total gains for each particular matrix size. Standard deviation shown in parentheses.

F Size	Algorithm	A' size	% Effective	Runtime (ms)
250x500	LU	346.6 (164.1)	53.3%	884.8 (205.0)
250x500	QR	250.0 (151.8)	100%	1057.6 (24.1)
250x500	SVD	250.0 (151.8)	100%	9989.5 (545.5)
500x500	LU	345.6 (164.0)	48.5%	689.3 (159.6)
500x500	QR	250.0 (151.82)	100%	702.15 (81.48)
500x500	SVD	250.0 (151.82)	100%	6931.0 (431.0)
750x500	LU	235.6 (35.4)	83.4%	577.5 (306.4)
750x500	QR	175.0 (56.6)	100%	452.7 (81.0)
750x500	SVD	175.0 (56.6)	100%	3170.5 (311.9)
50x100	Basic-MAW	60.9 (17.3)	53.6%	33304 (16694)
50x100	Add-OA	57.6 (16.3)	58.4%	28416 (20980)
50x100	Add-OR	48.6 (12.2)	73.1%	19622 (4780)
50x100	Add-OS	49.4 (11.8)	72.0%	9059 (3484)
50x100	Add-MAD	55.6 (16.4)	61.5%	35145 (13880)
50x100	Add-MAP	45.2 (11.8)	78.3%	2730 (733)
50x100	Add-MAW	52.6 (14.3)	66.5%	5190 (2035)
50x100	Add-MRD	87.6 (5.2)	18.4%	33409 (16544)
50x100	Add-MSD	49.7 (12.7)	71.4%	12675 (4488)
50x100	Add-MRP	87.9 (5.1)	18.1%	33475 (16116)
50x100	Add-MSP	87.9 (5.1)	18.1%	33145 (15982)
50x100	Add-MRW	87.6 (5.2)	18.4%	33424 (16580)
50x100	Add-MSW	99.8 (0.4)	0.3%	22610 (1681)
100x100	Basic-MAW	77.8 (16.5)	47.5%	38740 (14815)
100x100	Add-OA	77.1 (20.3)	41.4%	36493 (21889)
100x100	Add-OR	67.3 (20.7)	72.2%	26433 (15700)
100x100	Add-OS	66.7 (20.3)	74.1%	9924 (6624)
100x100	Add-MAD	77.3 (20.5)	41.1%	45602 (23751)
100x100	Add-MAP	64.6 (21.2)	79.6%	4170 (1737)
100x100	Add-MAW	71.7 (20.4)	58.0%	6975 (2265)
100x100	Add-MRD	96.5 (4.7)	6.0%	56812 (28320)
100x100	Add-MSD	67.7 (20.0)	72.3%	14727 (12835)
100x100	Add-MRP	96.3 (5.1)	6.2%	56416 (27898)
100x100	Add-MSP	96.3 (5.1)	6.2%	56453 (27972)
100x100	Add-MRW	96.5 (4.7)	6.0%	56754 (28308)
100x100	Add-MSW	99.6 (0.8)	1.5%	18984 (9136)
150x100	Basic-MAW	77.3 (15.5)	48.6%	44446 (18225)
150x100	Add-OA	78.0 (19.8)	40.8%	46702 (28646)
150x100	Add-OR	67.6 (20.4)	71.1%	39248 (25552)
150x100	Add-OS	65.8 (20.5)	76.1%	11672 (8922)
150x100	Add-MAD	77.9 (20.1)	40.6%	59397 (36990)
150x100	Add-MAP	64.6 (21.1)	79.9%	5788 (2554)
150x100	Add-MAW	71.9 (20.1)	57.6%	8694 (3107)
150x100	Add-MRD	96.7 (4.4)	6.0%	67981 (31839)
150x100	Add-MSD	67.7 (20.2)	73.2%	18458 (16314)
150x100	Add-MRP	96.5 (5.0)	6.3%	68117 (32045)
150x100	Add-MSP	96.5 (5.0)	6.3%	68173 (32209)
150x100	Add-MRW	96.7 (4.4)	6.0%	67929 (31739)
150x100	Add-MSW	99.6 (1.0)	1.5%	22105 (12201)