Methuselah: Intelligent Data Aging

Nick Lanham
Tim Kraska, Ed.
Michael Franklin, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-46
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-46.html

May 1, 2014

Methuselah: Intelligent Data Aging

by Nick Lanham

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Michael Franklin
Research Advisor

12/19/13

(Date)

* * * * * * *

Professor Ion Stoica
Second Reader

[illegible]

(Date)

12/20/13

Methuselah
Intelligent Data Aging

Nick Lanham, Tim Kraska, Michael J. Franklin

ABSTRACT

Recent years has seen an ever widening gulf develop between access times for data stored in memory versus data on disk. Concurrently, growth in main memory sizes has led to large gains in the popularity of database systems that keep their working sets primarily in memory. These systems make the assumption that either all data in always in memory, or that access to disk, managed by a standard buffer pool, will suffice.

However, with data sizes growing steadily and more quickly than available main memory, it is clear that all in-memory systems will need some way to move data to a cold backing store.

This paper proposes a new online, statistics based, batch-oriented technique to allow an RDBMS to leverage cold storage to increase data capacity without overly impacting query performance. Our solution couples well with semantic knowledge about an application, making it easy to take advantage of application specific access patterns. We develop a number of techniques for efficient statistics gathering and management, movement of data to cold storage, and querying of data in cold storage. We show this approach fits well into the main memory model, and that it has excellent performance in some industry standard benchmarks, as well as for an Enterprise Resource Planning benchmark we have developed.

1. INTRODUCTION

Recent growth in main memory sizes has led to large gains in the popularity of database systems that keep the entire database in memory [17, 16, 9, 13]. However, with data sizes growing steadily and more quickly than available main memory, it is clear that all in-memory systems will need some way to move data to a cold backing store.

It is therefore not surprising that several in-memory database projects have recently proposed aging techniques to move data from memory to disk. The H-Store team has proposed "Anti-Caching" [9], whereas Microsoft's Hekaton team has developed Project Siberia [13, 2]. Both systems can compute a set of infrequently accessed rows in memory and move those rows to some form of secondary or cold storage.

At its core, an aging system requires three main components: a method to identify which data is hot and which is cold, a way to move the cold data to secondary storage, and the ability to access the data in secondary storage when you need it. The previously mentioned systems are row-stores with a focus on high-throughput transactional workloads, and the point in the design space they have chosen for these components reflects that.

In contrast, our design focuses on column stores like Hana [17], and on enterprise workloads generated by systems like SAP ERP, which require a different approach. For instance, for our targeted workloads it is essential to be able to move data at a fine-grained level (e.g., attributes), a concern that has driven a number of our design decisions. At the same time, fine-grained tracking has more overhead; we require new secondary data structures used to manage cold data that do not occupy so much main memory they outweigh their benefits. For this reason, tracking the usage at the attribute level is nontrivial.

In this paper, we propose Methuselah, a column store providing fine-grained management over in-memory data. This is achieved by maintaining attribute-level access statistics over data in hot storage and by moving data to cold storage in batches based on those statistics. Our system can also leverage semantic knowledge about an application, making it easy to take advantage of application-specific access patterns.

Specifically this paper makes the following contributions:

- **Statistics Management & Compression** - We present low-overhead/low-footprint techniques for maintaining attribute level access statistics. These characterize the data access patterns of the running application and identify which data should be kept in hot storage, and good candidates for moving to cold storage.

- **Run-Time** - We identify and describe the necessary changes in the execution engine of a in-memory column store to cope with aged data. Our techniques are surprisingly non-intrusive and therefore integrate easily with existing column-store run-times.

- **Application Knowledge** - We present a method to easily integrate application specific information regarding column access patterns during aging, and show how this can improve performance.

- **End-to-End Analysis and Evaluation** - Methuselah is a full system. We consider how choices made at one level of the system effect other related components, as well as measuring and tuning these parameters. We evaluate our techniques using three benchmarks, one we have created, derived from real enterprise query workloads, and two industry standard ones.

The remainder of this paper is structured as follows. § 2 describes a typical workload and our proposed architecture. §3 describes how we track hotness at the attribute level using statistics. § 4 explains the use of those statistics to actually move data to cold storage. § 5 looks at how we leverage our batch oriented approach to build filters over the cold data. Finally § 6 covers our analysis of the performance of Methuselah as a whole.

2. TARGET WORKLOAD AND ARCHITECTURE

To motivate our design decisions, we first present a benchmark we have developed in an attempt to model more typical ERP workloads and walk through the execution of an example query in the system. We then describe the components in the system that are involved.

2.1 ERPBench

ERPBench is a benchmark we developed after analyzing the usage of various customers using SAP's Enterprise Resource Planning (ERP) software.[1] The benchmark aims to reflect typical ERP workloads by modeling both reporting and business transaction patterns, and using more realistic distributions for modeling how the users accesses items in the system.

ERPBench models a company with customers, suppliers and an available list of products. A simplified schema is shown in Figure 1, which shows all essential columns. However, ERP applications typically contain significantly more attributes per table. In fact, for ERP applications often "only 10% of the attributes of a single table are typically used" [17]. An example of how we reflect this is that our benchmarking schema has 40 columns in the Supplier table, although only 4 are actually used. Similarly, our OrderT table has 50 columns, but we only use 5 of them.

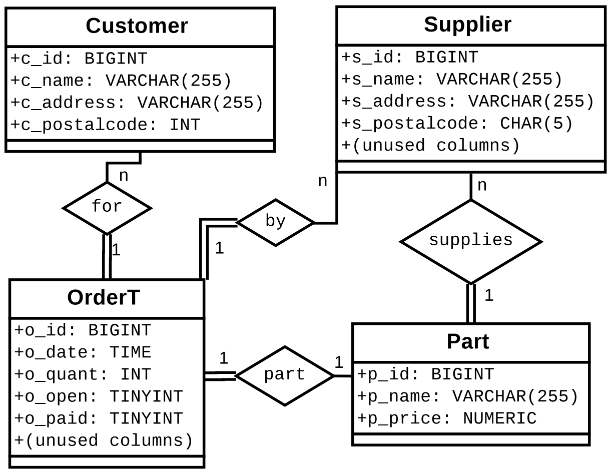


Figure 1: ERPBench Schema

[1]SAP generously provided us with anonymized log-records

We found that ERP queries are roughly divided into two access patterns: navigational/transactional queries and reporting queries. The former are fairly simple queries modeling typical business interactions, such us reading information about entities as would occur as a user moves around the company dashboard, as well as inserting new orders, customers, suppliers etc. For these queries we looked at user data to determine how much of the total data to model as "active" at any given time.

The second set of queries model reports that might be generated in a typical ERP system. Currently, ERPBench has two report queries: one that generates a yearly report of total sales, the best customer and best supplier and a report that lists all customers that currently have un-paid orders and the amount owed (a "dunning run") The query run for this report is:

```
SELECT c_name, c_address,
 (SELECT SUM(p_price*o_quant) as total from OrderT,
   Part WHERE o_paid=0 AND o_cust = C.c_id AND
   part = p_id) AS owed
FROM Customer as C
GROUP BY c_id, c_name, c_address ORDER BY owed DESC
```

2.2 Architecture

To support the above mentioned workload, we base our work on a Hana-like column-store architecture [17], which we extended with three new components (see Figure 2): The **Statistics Manager** (SM) receives, compresses and stores statistics generated during query execution. The **Aging Manager** is responsible for taking statistics gathered during normal operation and applying a policy to generate the list of data items that should be moved to cold storage. The **File Access Module** uses filters to attempt to avoid going to cold storage, but can also read cold data when absolutely needed. Finally, we modify the **Query Executor** (QE) to talk to the File Access Module and report statistics to the SM. We discuss each of these components in detail in the following sections.

2.3 Example Execution

To provide an overview of how Methuselah executes queries, we step through the execution of the example query shown below. The details of the filters are described in § 5, but for now simply consider them something we can probe to determine if a value exists on cold storage. The query we consider is one of our simple dashboard queries, this shows the number of orders today for customers near me (useful for a summary screen in the dashboard)

```
SELECT COUNT(*) FROM Customer,OrderT
WHERE c_postalcode = $codearg AND OrderT.for = c_id
 AND o_date = CURRENT_DATE;
```

As a first step we push down the where conditions to the columns where possible and scan the memory to retrieve all relevant hot-data rows (step 1 in Figure 2). While scanning, statistics are also recorded. We then need to check if it is necessary to query cold storage. For instance, in generating the list of row ids from the OrderT relation, we will probe the filters for the current date, and if it is not contained in the filter, we can avoid going to cold storage for the OrderT relation. Likewise, we check the filters for the specified postalcode, to possibly avoid reading cold storage (step 2 in Figure 2). If, on the other hand, the filters indicate there might be

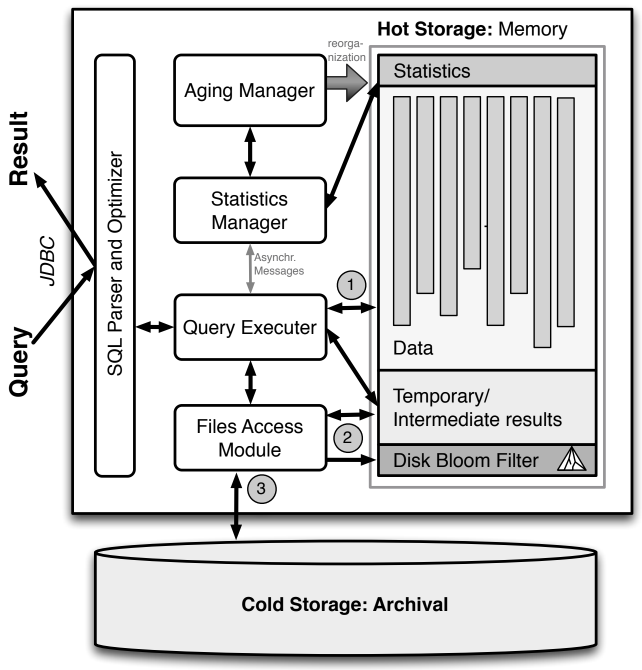


Figure 2: System Architecture

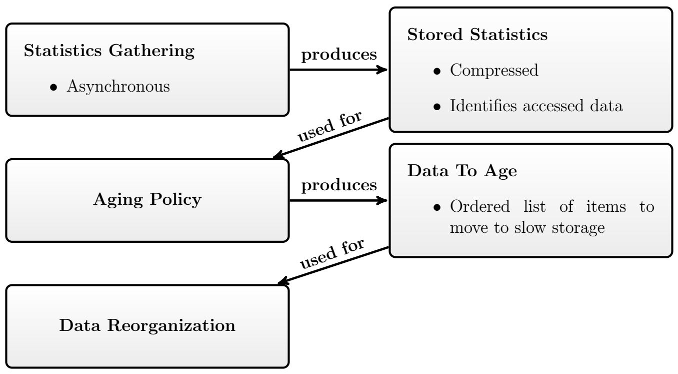


Figure 3: Aged Data Generation

cold data to read, we simply put a place holder in the buffer and read the data lazily when it is actually needed, in the hope that some other part of the query will narrow down the needed rows (step 3 in Figure 2). This generated data is placed in a temporary buffer associated with the query.

We now need to perform the join. Using an estimate of the cardinality we would generate the list of customer ids to check for by either reading the OrderT or Customer table. If either can generate the list without reading cold data, it is used. We then join against the other relation by scanning for matching ids. Here again our filters can be very useful. They can tell us, for instance, that none of the orders we are interested in have data in cold storage, meaning the join does not need to scan any cold data.

If the filters indicate that there might be data in cold storage we need to read, the File Access Module scans for the required data and places it in a temporary buffer with the other partial results. Finally the number of matching rows is counted, returned the user, the and the temporary buffer is reclaimed.

2.4 Aging Process

An overview of the aging process is shown in Figure 3. During query execution we gather statistics about what hot data is being accessed (§ 3) and store these statistics, compressing them significantly to reduce overhead (§3.3). Using the statistics, an aging policy decides which set of data should be aged (§4). Finally, the selected to be aged data items are moved to cold storage. During this process we also generate filters over the cold data (§ 5), to minimize the impact of aging on the run-time.

3. STATISTICS

The first step in selecting data to age is having methods to characterize attribute-level access patterns via statistics. Item (attribute) level patterns are required for two reasons. First, typical enterprise systems have many columns [12], and access frequencies can vary greatly on both column and row dimensions. Second, as [9] notes, if hot items and cold items are bound together, either by row or page, the hot items can force the cold ones to remain needlessly in memory.

To maintain performance in the active system these collection methods must impose a low overhead during query execution, not consume a large amount of storage, and must characterize access patterns without a need to query the active system. In this section we describe methods for gathering, storing, and compressing statistics consistent with these requirements, and why we have opted for these methods over possible alternatives. We show the memory overhead of our techniques in § 6.6.2, grouped with our other experimental results.

3.1 One Bit Statistics

We have designed Methuselah's statistics subsystem to provide efficient performance, sufficient characterization of data access patterns, and to place minimal load on the active system.

To minimize overhead during query execution we decouple access of an item from the recording of that access. When the Query Executor access an item it sends an asynchronous message to the Statistics Manager (SM). This "fire-and-forget" model means the Query Executor does not need to wait for the SM to actually record anything, and can run queries at full speed (note, that for large scans, we can further minimize the required number of messages as explained later.)

When the message is received in the SM we store only a single bit per item accessed. If the bit is set for an item it indicates that since the last aging run, that particular item was accessed. This approach has two major benefits. Firstly, we reduce the amount of memory needed to store the statistics by storing only one bit (§ 6.6.2 quantifies this benefit). Secondly, for any item accessed more than once, subsequent messages generate only a read in the SM (as the bit does not need to be re-set), making the component more efficient. To further reduce the size of the stored statistics we have developed a number of logical compression techniques, detailed below.

The downside of our single-bit approach is that we potentially mark too much data as "hot" between aging runs, and that we have no way to reason about the relative hotness (i.e., we only know if the data was accessed or not). Surprisingly, as shown by our evaluation, for many scenarios this simple statistic is sufficient, if used together with sampling and intelligent aging ordering as described in the following sections.

Another method would be to simply log every query executed, and no statistics about items at all as done by Siberia [13]. This approach is efficient at query execution time, requiring only a single log entry per query, however, during an aging run, the system would need to re-execute every query to determine which data items were accessed and which were not. While a technical possibility, this places a large overhead on the active system during aging time and would only be feasible if the active system were mostly underutilized.

3.2 Statistics Sampling

One downside to storing a single bit per attribute is that a query that touches almost all data can mask which items are more frequently accessed and which where only touched once or twice by a large scan. We use a simple sampling strategy to overcome this effect. Unfortunately, sampling also has the effect of not recording many items that are only accessed a few times in a period, which we study in more detail in the experiment section § 6.6.

Sampling the messages also justifies the "fire and forget" model in the Query Executor for statistics messages. Because we actually want to drop some number of messages, we need not be concerned about the reliability of the messaging channel.

3.3 Statistics Compression

Keeping a statistic per attribute access can lead to very high storage requirement for the statistics, and we have therefore developed techniques for reducing the overhead of storing statistics. The idea is to store logical information about what items were accessed, rather than information about every item. In the following sections we describe a logical compression scheme which significantly reduces the overhead of the SM. An empirical study of the effectiveness of this logical compression is shown in § 6.6.2.

In contrast to the query logging alternative discussed previously (actually an extreme form of logical compression), this approach can determine if an item was accessed without needing to invoke the runtime system. All required information is in the SM, keeping the overhead minimal.

3.3.1 Simple Logical Compression

The simplest logical compression technique is for storing access information for a query such as:

```
SELECT AVG(salary) FROM Users
```

where there is no where condition, and we must clearly access the entire salary column. In this case the Query Builder sends just a single "scan" message to the SM indicating that the entire salary column was accessed in this period. This column will then be marked as "scanned" in the SM for age set determination. (See § 4.2 for how this is used during aging).

Queries in real applications tend to not be so simple however. Moving up one level of complexity we examine the following query:

```
SELECT AVG(salary) FROM Users WHERE salary > 15000
```

We can tell statically in the Query Builder what items will logically be accessed, and send a single message with the column and the WHERE condition, which will be stored in the SM. When running aging, we can tell if an item was accessed by checking if any of the stored where conditions match the item, and, since the conditions are stored in the SM, we do not need to actually execute any queries.

3.3.2 Advanced Logical Compression

The above techniques give some reduction in statistics size, but provide no benefit in the face of more complex queries, for instance those with joins or projections. In this section we describe techniques to reduce statistics size in the face of joins, by modeling them as WHERE clauses. We also compress statistics on selected columns without WHERE predicates, by remembering which other columns in the query *did* have a predicate applied.

To understand these techniques, consider the following query, which selects all employees in Germany, and the name of their departments. In addition each department is responsible for a number of products and we select the average price of the products that department is responsible for.

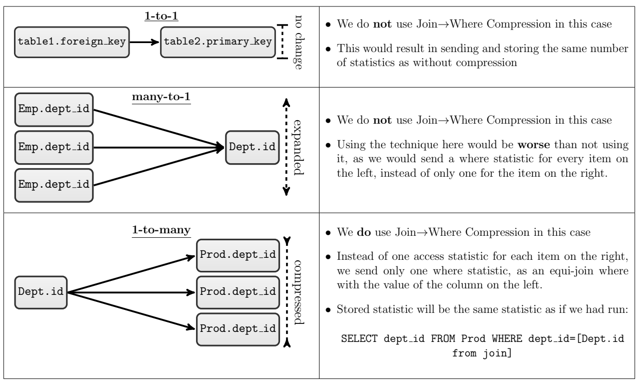
```
SELECT Emp.name, Dept.name, AVG(Prod.price)
FROM Emp, Dept, Prod
WHERE Emp.region='de' AND Emp.dept_id=Dept.id
 AND Dept.id=Prod.dept_id
GROUP BY Emp.name, Dept.name
```

Using only what we have described so far, in this case, we will compress the access to `Emp.region`, but we will not compress access to any of the other columns involved in the query, and each item accessed will be stored as a new statistic. To improve this situation we need two new compression techniques:

1. Join→Where Compression
Since each department could be responsible for a large number of products, the join condition above will result in lots of statistics being recorded, one for each product that an employee in Germany might be responsible for.

If we rather think of the join as the application of a number of `WHERE` conditions against the `Prod.dept_id` column, we can compress all those accesses and rather record a single where statistic for each department considered.

This form of compression is only beneficial if we are performing a 1→many join, as in `Dept.id = Prod.dept_id` above. Figure 4 shows the various join cases and when this technique can be used.



- We do **not** use Join→Where Compression in this case
- This would result in sending and storing the same number of statistics as without compression

- We do **not** use Join→Where Compression in this case
- Using the technique here would be **worse** than not using it, as we would send a where statistic for every item on the left, instead of only one for the item on the right.

- We **do** use Join→Where Compression in this case
- Instead of one access statistic for each item on the right, we send only one where statistic, as an equi-join where with the value of the column on the left.
- Stored statistic will be the same statistic as if we had run:

```
SELECT dept_id FROM Prod WHERE dept_id=[Dept.id
                 from join]
```

Figure 4: Use of Join→Where Compression

2. Selected Column Where Compression (SCWC) In the above query we access a number of columns that have no `WHERE` condition applied to them, but with column access nevertheless depends on the result of a `WHERE` condition being applied to a different column. This means, for example, a statistics bit will need to be stored for the `Emp.name` column for every employee in Germany.

To compress this access, we note during query planning the columns accessed that are dependent on the `WHERE` conditions applied to other columns, and record that as a single statistic. This can lead to a significant reduction in total statistics as filtering on one column and projecting another is a very common operation.

Figure 5 shows the relationship between the column that has a `WHERE` condition and statistic we will store. The solid line is used during query execution and at aging time, to determine which rows to access and which rows were accessed respectively. The dashed line shows the data that is stored in the SM, namely that `Emp.name` was accessed with a `WHERE` condition on `Emp.region`. We store this once, rather than for each matching item in `Emp.name`

Currently for the above query we would compress the accesses to `Emp.name` and `Emp.dept_id`, but not to `Dept.name` or `Prod.price`. Figure 6 illustrates the re-computation work that would need to be done in order to store a compressed statistic. While this is technically possible, it places an undue burden on the active system during aging. The gray arrow indicates the compressed statistic that we could store, but actually do not.

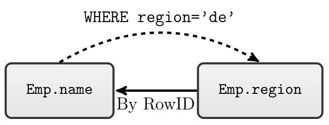


Figure 5: Same Table SCWC

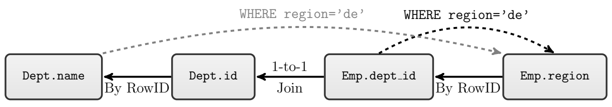


Figure 6: Joined Table SCWC

Note that, for both of these techniques, we have all the information needed to reconstruct access in the SM, and have no need to burden the Query Executor to determine aging candidates.

4. AGING POLICY

Following Figure 3, we now use the statistics from the previous section to determine which data to actually age (the "age set"), and when to age it. In this section we develop a number of heuristics to rank in-memory data, to choose exactly which data to move to cold storage, and for when to trigger an aging run.

The policy described here makes a number of choices, each of which has various parameters that can be tuned. We explore a number of possible parameters settings and present the results in §6.

4.1 Free Memory Target (FMT)

When aging, Methuselah must choose how much free memory will be available at the end of the run. This choice effects not only how well the aging approach will be able to avoid going to cold storage in the future, but also how often aging will need to be run.

If Methuselah tries to free too much memory, then frequently used items will be moved into the age set, leading to poor performance. On the other hand, if too little memory is freed, then aging will need to be run frequently.

Overly frequent aging can also cause the statistics to provide a poor characterization of the actual access patterns. If aging is too frequent, the actual working set of the system might not be captured, and items that should not be aged out will be. This effect is seen in the AuctionMark benchmark, and is analyzed in §6.

The optimal value for the free memory target is highly dependent on workload. The lower the insert rate, the less free memory will be needed after a run. Methuselah uses a heuristic to move the target between some limits, which we have found to produce the best performance.

We currently use an upper limit of 30% of memory to be free after aging. We have found that more than this poor performance on all workloads we have tested. Furthermore, if the workload is such that 30% of memory is being filled in a single day, then an in-memory system is probably not well suited to the use-case.

We use 5% as a lower limit. Any lower risks leading to overly frequent aging runs in the face of a slight increase in insert rate.

Methuselah currently self-adjusts between these limits to have aging run about once a day[2]. If aging is triggered too early, the FMT is increased by an amount proportional to how early aging happened, and likewise the FMT is reduced if aging occurs after more than a day. We found this provides a good trade-off between performance and frequency of aging.

4.2 Ranking

Based on the gathered statistics hot data is ordered by a series of criteria. Items are initially separated into five sets:

1. **Untouched** - Items that have been neither accessed nor scanned in the most recent period
2. **Scanned** - Items that were only scanned, but not accessed (accessed meaning they never became part of the result set of the query)
3. **Accessed** - Items that became part of the result query
4. **Updated** - Items that have been updated

Untouched items are the best candidates for aging, as they did not participate in any queries. At first blush, it might seem that scanned items are not a good choice, but the filters we describe in §5 will likely prevent us from having to read these items from cold storage in the future. Finally, updating items in cold storage is expensive and so updated items are ranked last.

Each set is sorted by item size and then, considered in the order given above, items are added to the age set. Once enough items have been added to hit the free memory target, aging is run.

[2]in our benchmarks we define a "queries per day" amount to simulate this

4.3 Application Specific Knowledge

When choosing the set of data to age we can also leverage application specific knowledge to make better decisions. By using knowledge of the inner workings of an application, developers can provide better recommendations than the statistics regarding which data should be aged. We have made an effort to ease the integration, as many applications already have a large corpus of such knowledge.

Methuselah supports three types of application specific knowledge: A list of columns that should not be aged, queries that select items that should be aged, and finally columns for which the ordering is an indication of the suitability of aging.

No-age columns: No-age columns can be useful if a developer knows a column is frequently accessed, but also that the pattern is such that the statistics will not provide a good indication of what to age, or where developers have a specific query in mind that must always be fast, and therefore want to enforce that the query always remain in memory.

For example, in ERPBench the modeled dashboard has the ability to query parts by a user supplied name, and based on the data we looked at, there is little to no pattern in the searched parts. We therefore mark `Part.Name` as a no-age column to ensure this query is always fast.

Aging queries: A more flexible type of application knowledge is a set of queries that can be applied at aging time to identify data that should be added to the age set before data that would be selected by the heuristics above. For example, an application might know that rows in the order table for which the status is 'closed' will almost never be accessed. Methuselah can therefore move that data out eagerly.

We require that the queries specified for aging not perform any joins. This is so it can be determined if a data item matches the query without placing a burden on the query execution engine. For the same reason the aging queries cannot perform aggregates or group bys.

Aging order: The final kind of application knowledge supported is a method to indicate that a columns ordering provides a hint regarding suitability for aging. For example, an app developer might know that recent items are always accessed more than older items for a particular table. In this case, the date column could be specified as an age-ordering column, and the order would be applied along with the ordering constraints given in the previous section.

Applying aging order gives rise to a number of subtle problems. For instance, when should the developer specified aging order take priority over the statistics order. For now we implement only a simple form of aging order where the developer always wins, and consider more advanced cases as future work.

4.3.1 ERPBench

ERPBench (§2.1) takes advantage of the ease with which Methuselah can integrate application specific knowledge. Specifically we provide the following specification:

- `Part.name` is a no-age column
- `Order.date` is given as an aging order column
- `SELECT * FROM Order`
  `WHERE Order.closed=1 AND Order.paid=1`
  An aging query that specifies that paid and closed orders (which are rarely queried), are good candidates for aging.

Table 1 shows that we keep about 12% more queries in memory thanks to this specification.

5. FILTERS

Due to the fact that we move data to archival storage in large batches, we have an opportunity to build compact filters over the data, reducing the times we need to access the data in the future. This section describes the filters we build. The memory overhead we incur to store them and the performance benefits from the filters is show in §6.5.

5.1 Bloom Filters

When a query requires some data that isn't in hot storage, data should only read from cold storage if it is actually there. Normally one would need to scan the data to find a requested item, but since the system knows exactly which data is being moved to cold storage, a compact bloom filter [3] can be built to allow aborting early if the data is definitely not in cold storage.

5.1.1 Sizing The Filters

In order to keep our bloom filters small, we use a heuristic algorithm to size them. Initially the filter is limited to only use `1Kb`, and as long as we don't see any false positives from the filter, it will remain small. As soon as we get a false positive however, two things happen. First, a hash table is built to avoid ever reading that item again, and secondly, a false positive counter is incremented. If the counter gets too high (over 100 in our experiments), a re-build of the filter is triggered in the background.

This scheme works well for two reasons. Firstly, we have found that the cardinality of the sets of false positive items is quite low, and therefore the hash table never grows very large. Secondly, there are many columns that themselves have low cardinality, and therefore can be well represented by a very small bloom filter.

To determine the optimal number of hash functions the technique described in [6] is used.

Using the above techniques our bloom filters never take up more than 4% of Methuselah's total memory. The exact fractions are shown in Figure 8.

5.2 Data Characteristics

In addition to the bloom filters, min, max, sum and count values for the aged data are maintained. These values allow the answering of aggregate queries more efficiently, and also avoid scanning on range predicates, if the desired range falls outside the range of the aged data.

We are aware that these "grouped" values are simplistic, and that there is a significant amount of work on materialized view maintenance that could benefit our approach here. We see this as an exciting direction for future work.

6. EXPERIMENTS

This section shows our end-to-end evaluation of our Methuselah system using three different benchmarks, TPC-W, AuctionMark and our own ERP benchmark (ERPBench).

In the remainder, we first describe how we evaluated the system and the benchmarks. We then present the overall performance results (§ 6.3), study the effects of our filter implementation (§ 6.5), and finally examine in detail the effects of the various parameters of the SM component (§ 6.6).

6.1 What To Measure

In a system with data in cold storage, average query performance will be dominated by any queries that need to touch cold data. Similar to a CDF, Figure 7 (note the log scale) shows the average performance by including fractions of the queries pre-sorted by their response time.

For queries that remain in memory, very little time is spent reading data, but as soon as a query needs to read from archival storage, the runtime balloons enormously. The knee in the graph at 87% is because we keep that fraction of queries in memory for this benchmark. Since the difference is so great, mean query performance (the upper dotted line) is completely dominated by the cold queries.

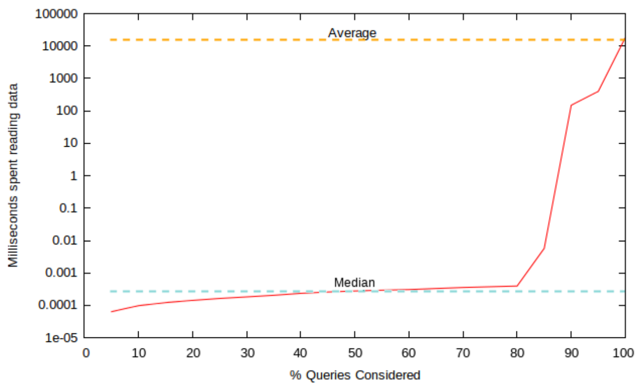


Figure 7: Reading data in AuctionMark

We therefore argue that it makes sense to consider in-memory queries separately from archival queries and that the fraction of queries that remain completely in memory is the appropriate figure of merit in this setting.

Furthermore, the performance gap means that any extra information that can prevent an access to cold storage is a huge win for the system. Our batch oriented aging approach means we can build a summary of the data, and efficiently answer certain aggregate queries, as well as filter queries that would have otherwise had to touch cold storage. §6 quantifies the benefits of this approach.

6.1.1 Baselines

In order to compare our aging strategy, we also implemented LRU and LRU2 [11] caches in Methuselah. When running in these modes Methuselah moves data in and out of cold storage on demand at the page level, and records the fraction of queries that incur a cache miss as the fraction of cold queries.

In addition, we have implemented a similar algorithm used for “Anti-Caching” [9]. In this case, we maintain an LRU chain for whole rows as they are accessed, and move rows to cold storage when we cross a memory threshold.

As would be done in a normal system, for the caching modes we build an index over key columns and any other columns that specify an index in the schema. These indexes occupy main memory space, but also are used to prevent cold reads when possible.

We report the fractions of queries that remain in memory in these modes in Table 2.

6.1.2 Query Replay

To ensure we compare each configuration on exactly the same workload, we have built a replay mechanism into the system, so we can replicate the exact query execution for each configuration to test. There is a mode that dumps every query received via JDBC to a text file. A “reference” run is created by executing in this dump-to-file mode. We then run our tests of the various configurations of interest by reading and executing the queries from the reference file.

6.2 The Benchmarks

We use two industry standard benchmarks, and ERPBench, a benchmark of our own design. We described the details of ERPBench in section §2.1. For ERPBench we used a ratio of 1:10 of hot to cold storage. That is, the system has to keep 10 times more data in cold storage than in-memory.

In addition to ERPBench, we also used variants of AuctionMark and TPC-W.

6.2.1 AuctionMark

The AuctionMark benchmark[1] models an OLTP system similar to a well-known auction site. It involves 13 tables and 14 specific procedures that give a good representation of the site’s core transactions. We use the default transaction mix of AuctionMark, with a time target of one hour. For this benchmark we used again a 1:10 hot to cold ratio.

6.2.2 TPC-W

The TPC-W benchmark[14] simulates a transactional e-commerce website. It involves a number of interactions that model a business oriented transactional web server We have implemented a TPC-W like workload the runs all the queries from all the TPC-W interactions. The reference run is created by running 300,000 interactions. We run transitions according to the “Ordering Mix”. This mix is the most write and insert-heavy workload and therefore the best stress-test of our aging algorithms. For this benchmark we used a 1:8 hot to cold ratio as a smaller ratio (e.g., 1:10) turned out to force almost every query to disk.

6.3 Results

For evaluating our system we tested the following settings:

- **Full Aging** - This is our full solution. Dynamic FMT, cold columns, and in the case of ERPBench, application specific knowledge. This represents the best Methuselah currently performs for these benchmarks.
- **Statistics Dynamic FMT (Stats Dyn)** - This uses all the statistics tricks, but neither marks cold columns, nor uses application specific knowledge.
- **Statistics 10% FMT** - Fixed FMT of 10%
- **Statistics 20% FMT** - Fixed FMT of 20%
- **Statistics All Eligible (All Elig)** - Simply age out all data that was not accessed in the last aging period.
- **Statistics No Filters (No Filter)** - This method does not use any filters to reduce cold access
- **LRU2/LRU** - Traditional caching models as described in §6.1.1
- **Whole Row LRU** - Similar to [9], we track LRU rows, and move data at the row level.

Table 1: Hot Queries: Technique Comparison

Benchmark	**Full Aging**	Stats Dyn	10% FMT	20% FMT	All Elig	No Filter
ERPBench	**92%**	80%	79%	75%	68%	63%
AuctionMark	**87%**	79%	71%	76%	70%	37%
TPC-W	**93%**	91%	90%	87%	85%	53%

Table 2: Hot Queries: Caching Comparison

Benchmark	**Full Aging**	LRU2	LRU	Whole Row LRU
ERPBench	**92%**	65%	70%	58%
AuctionMark	**87%**	71%	66%	61%
TPC-W	**93%**	84%	80%	74%

For all benchmarks, we measure the fraction of queries that stay entirely in memory, referred to as "hot queries". **Hot queries** are the number of queries that touch only hot storage, and are an indication of the overall "speed" of the system, because of the significant speed difference between hot and cold storage. If not stated otherwise, all results use a sample rate of 80%, as we found it to achieve the best trade-off between overhead and precision throughout all the benchmarks.

The results for the different settings of Methuselah are shown in Table 1, whereas Table 2 shows the hot queries numbers of Methuselah's best case performance against the various caching strategies.

6.4 Analysis

6.4.1 Overall Result

The results show clearly that our full aging solution significantly outperforms the various caching techniques in keeping a high percentage of queries purely in hot memory. Our full aging approach helps to keep 9% to 22% more queries entirely in hot storage without ever touching cold data.

Even more striking is the difference when whole rows need to be moved out together. While this is not as critical for AuctionMark and TPCW, in ERPBench, which has the typical wide rows seen in an ERP system, it causes a 12% drop from attribute level LRU, and a whopping 34% in hot queries from our approach.

Our solution performs better for different reasons for each benchmark. For ERPBench, the reporting queries cause the caching modes to evict needed data, leading to lots of cold reads, but our statistics see this data as only scanned, meaning it is a better candidate for aging. This means the more frequently accessed hot data remains always in memory. In AuctionMark and TPCW it is the filters that produce better performance, as there are a lot of point queries, going to cold storage can often be avoided. In addition cold columns help a lot in the AuctionMark benchmark for reasons outlined below.

A general trend we have found is that the most frequently run queries are not the queries that tend to run slowly. The reason is twofold. Firstly, in real systems it is often the case that queries that run a lot are simplified to run quickly, meaning they don't touch a large amount of data. Secondly, as these queries run often our aging algorithm doesn't pick the columns to be aged, and therefore the whole column tends to stay in memory.

The queries that tend to go slowly are rather those that are run a medium amount and that access quite a lot of data. Queries like, get all this customer's information, or, get all this users comments, fall into this category. The data these queries hit is not as hot as for high frequency queries, so some items they might access (but were not in a particular aging period) are aged out, and then when they are needed, quite a lot of data has to be read from slow storage.

Surprisingly, for all benchmarks, the the effect of the dynamic threshold compared to the fixed ones only helps to improve the *hot queries* by 3-4 percent. However, cold columns has a huge impact of up to 12 percent for hot queries. This leads us to believe, that it is actually more important to determine the right order of what is aged, instead of how much to age per aging run.

Looking at the impact of the different optimizations, it can be noted that not every optimization helps with every benchmark. For example, adding filters (i.e., No Filter to All Elig) improve the *hot queries* percentage for AuctionMark and TPC-W significantly, but are less important for ERPBench. Below, we break down some of the reasons for these differences.

6.4.2 ERPBench

For ERPBench the majority of the slow queries are from generating the required reports. These queries need to scan over a large portion of the data space, but are not run extremely frequently, and therefore tend to hit cold items. Important to this benchmark, however, is the the queries pass over the same items more than once in generating the reports, so LRU2 provides the worst performance, with 4% fewer hot queries than LRU. Whole Row LRU performs very poorly here, as this benchmark has some very wide, as is common in ERP schema. This causes a significant amount of cold data to need to remain in memory, hurting performance.

The active set of data for the dashboard queries means they can be kept mostly in hot storage. We see that the ordering techniques provide about 10% more hot queries in the aging case, as the items not actually accessed (i.e. old orders) are aged first. The reporting queries disrupt the caching algorithms however, causing them to throw out most of the data needed for these queries and thereby leading to poor performance for a time after the reports are generated.

These queries can effect the aging results, however, we leverage our app specific knowledge to influence aging order via date columns. This means that even if, say, the majority of orders in hot storage have been accessed, the system will still favor aging older orders, leading to better performance in the aging case. This is possible due to the batch oriented approach taken, and can't be taken advantage of in the caching modes. If we run our aging solution without our app knowledge, only 73% of queries are kept in hot storage.

The filters play a lesser role in this benchmark as most of the queries scan for items and do not perform direct access.

6.4.3 *AuctionMark*

The AuctionMark benchmark is split into various "transactions". We run the standard mix specified by the benchmark. There are two transaction types that account for the bulk of the slow queries in this benchmark.

The "GetItem" transaction is run 45% of the time and illustrates the importance of getting the FMT correct. By dynamically adjusting we are able to keep 8% more queries in hot storage. This transaction is run only on "open" auctions (those auctions which have not yet ended), meaning old items are not touched. There is therefore a working set of active items, those in open auctions, at any given time. If the FMT is too low aging is run before the majority of working set is accessed, and end up aging out some still active items. This leads to more slow queries and reads.

The other transaction that represents the remaining bulk of cold queries is "GetUserInfo". This transaction retrieves a substantial amount of data about a given user, and since the user it is run on is random, it is more likely that aged/uncached data is touched. These queries plus the "GetItem" queries make up about 75% of all slow access.

The "GetUserInfo" queries also explain why AuctionMark doesn't perform as well as TPC-W and why we get the 31% bump in hot queries from our filters. The extra data read by this transaction both brings down the percentage of in-memory queries, and increases the number of aged reads. That being said, many of the accesses can be filtered out, in the case that say, a user has not made any comments on an item. Without our filters we always have to check cold storage for this data.

The approximately 10% increase in hot queries (SAE to the FMT/SDF modes) comes from our ordering techniques. The "GetUserInfo" queries touch only a subset of all columns, which are marked as Accessed and Scanned appropriately. These columns are then lower in the order generated during aging, and more of their data remains in hot storage, keeping many of the "GetUserInfo" queries fast.

It should be noted that the picking of a random user is probably somewhat unrealistic for a true auction system. One would expect that some users are more active than others, and that their "GetUserInfo" queries would be run more often. In this case Methuselah would perform even better as those users would not be moved to slow storage.

The reason that cold columns provide an 8% increase in hot queries in this case is two fold. By reducing the amount of aging runs necessary, the system gets more of the "auction working set" benefit described above. Also, more frequently accessed data can be kept in memory, as there are a number of fairly large columns in this benchmark which are never accessed.

6.4.4 *TPC-W*

The TPC-W benchmark has three main interactions that cause slow queries. These are, the Product Detail Web Interaction, the Buy Request Web Interaction, and the Search Result Web Interaction. This is unsurprising as they are the most likely interactions that access any significant data in the mix we run for TPC-W.

There are a number of unused columns in the benchmark. For example, `author.a_bio` and `author.a_mname` are both rather large columns that are never accessed. As new data comes into the system, having these columns be cold columns gives us 2% more hot queries, as more relevant data can be kept in memory.

These unused columns also explain why the Whole Row LRU strategy performs poorly. The effect is not as dramatic as in the ERPBench case, as there is not as much unused data in this benchmark.

Both the Product Detail and Buy Request interactions select a significant amount of data with a random predicate. For Product Detail, the product retrieved is random, and for Buy Request the user making the request is. This means there is a higher chance some of the relevant data has been moved to slow storage. Again, this is somewhat unrealistic for a real system, and we would expect somewhat better performance on a non-synthetic workload, and again these queries are where the filters provide us 30% more hot queries.

It might be expected that the Admin Confirm Interaction would generate a lot of slow reads, as it potentially touches a lot of data, and indeed it does contribute some slow reads and scans to the final count. However, these queries are issued relatively rarely they do not contribute a significant amount compared to overall data volume.

Note that the most frequently run queries are not the queries that have to read cold data. The most run query for TPC-W grabs the latest cost for a particular item. However, this query needs only the id and cost columns from the item table. As these columns are accessed so much these columns are kept entirely in memory.

6.5 Filters

§ 5 explains how we build filters to minimize the times we need to access cold data. In this section we vary how large we allow these filters to grow, and in Figure 8 report the fraction of hot queries. These numbers are for the ERPBench benchmark, but we saw similar results for the other benchmarks.

The hash table (see §5.1.1) is clearly a valuable component of the filter system. It saves us on about 10% of queries. Additionally, this table remains very small, never exceeding a fraction of a percent in our tests.

We also observe that limiting the size of the filters is important. Letting them grow to more than about 4% of memory size has a detrimental effect on performance. The fraction was similar for our other benchmarks.

6.6 Statistics Experiments

We used the AuctionMark benchmark to measure the effects of sampling, and the memory overhead of the statistics we store.

6.6.1 *Sampling*

The effect of our statistics sampling approach is shown in Table 3. It might seem counter-intuitive that 50% of the

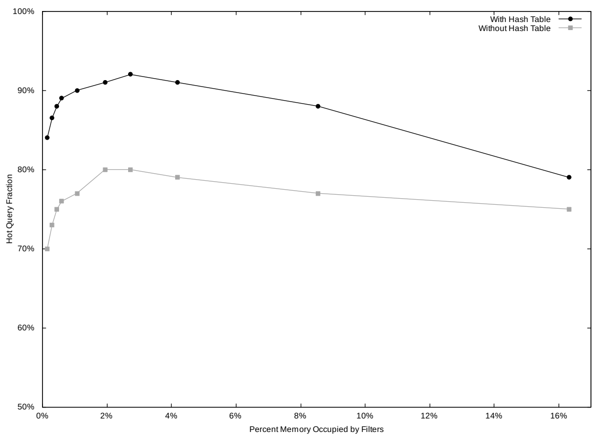


Figure 8: Effect of Filter Size on Hot Queries

messages sent to the SM could be dropped, and that performance could remain so good. The reason behind this is that a large number of redundant messages are sent. On average, 43,809 unique statistics are maintained, while an average of 5,241,394 messages are sent to the SM. That means, if we got very lucky, 99% of the messages could be dropped and still have perfect information. This also illustrates why statistic compression is so important.

[13] also uses sampling in their approach, and found it provided an accurate characterization of the hot set of data.

Messages Dropped	In Memory Queries	Slow Reads
%0	86%	0.96%
%10	87%	0.96%
%20	87%	0.97%
%50	88%	1.01%

Table 3: Performance with sampled statistics

6.6.2 *Statistic Size Experiment*

In order to test the statistics compression techniques we have developed, we have run a number of experiments on the size of the statistics needed. For these experiments we use ERPBench, and also report LRU as a baseline measure.

For all statistics based results a FMT of 20% was used. For LRU we compute the average size of the LRU list over the entire run, and assume a 4-byte per entry data size.[3]

Below we report the percent of total memory needed to maintain the data structures for statistics and LRU. The **Stats No Comp** bar indicates the size of statistics with no where compression at all. **Stats WC** only compresses accesses to columns that have a `WHERE` condition applied directly too them. **Stats WC+JC** uses both of the previous techniques, plus Join→Where Compression. Finally,**Stats WC+SC** is all the above, plus Selected Column Where Compression.

[3]This is actually rather generous to LRU, since it is normally implemented as a list plus a hash-table for fast lookup, and would therefore require more than 4-bytes per item of storage overhead

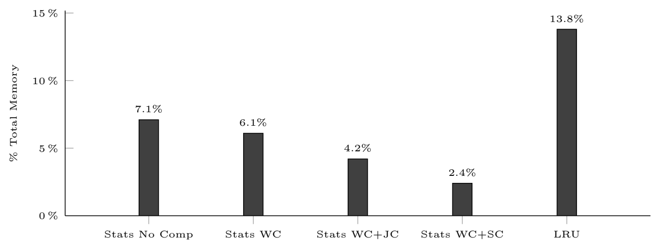


Figure 9: Percent Memory Used by Statistics/Cache

7. RELATED WORK

We focus our efforts on primarily in-memory column oriented data stores. [4, 5, 17, 18]. Our techniques are not directly tied to any specific system, however, and could generally be applied to any system with attribute level access that would like to avoid traditional caching systems, or has a workload that tends to have significant quantities of largely unused data.

Microsoft's Project Siberia has tackled a similar problem in the context of the Hekaton in memory OLTP engine [10]. [13] presents a method to identify hot and cold records in a main-memory database, based on scanning logs of record access generated during normal operation and [2] presents Adaptive Range Filters (ARF), a novel filter targeted at range queries, used where Methuselah uses bloom filters.

[9] presents "Anti-Caching", another approach to supporting moving data to slower storage. The system, however, is closely tied to the architecture of the underlying system (H-Store), and also does not consider that data might be on much slower storage than a local disk.

Both the above approaches are row-stores with a focus on high-throughput transactional workloads. As such, they have made different design choices from our system and target different workloads.

ARFs are something Methuselah could benefit from, however, as we show in §6.5 it would be important to quantify the trade-off of filter size vs. the benefit provided.

The Oracle TimesTen database provides a form of data aging [15]. This can be either LRU based or user-defined, but for TimesTen, "aged" actually means deleted. That is, TimesTen does not move aged data to slower storage and continue to provide access to it. Rather, it simply deletes items based on the specified policy.

HP AutoRaid[19] moves data automatically between fast and slow storage based on access patterns. This work is focused on file-system performance, however, and therefore cannot leverage semantic information about the data being stored as we do in building the various filters over our data.

Semantic Caching[8] showed that better cache performance can be achieved by taking into account the semantics of the predicates being applied to queried data. Our Where compression (§3.3) techniques are an application of this idea in a different context, and at aging time, the stored where statistics act partly like a semantic cache.

There has been a wealth of work regarding optimal caching algorithms[7, 11]. Our analysis shows however, that due to the number of attributes that need to be tracked, they are not as well suited to our target workloads.

8. FUTURE WORK

We have developed a number of new techniques for managing hot and cold storage for this work. There are many ways in which we can improve on the techniques described here.

As noted earlier, most in-memory column stores use compression techniques to increase the amount of data that can be kept in memory. We would like to model this more carefully, as we feel it will be a benefit for our system.

For example, we do not currently model the benefit of reducing the cardinally of a column due to aging, but if we can reduce the number of bits needed for the index vector, we could see significant gains. This will also effect the ordering chosen when generating the list of data to age, and these effects will need to be measured.

Certain columns provide an excellent indication of the suitability of a row for aging. Examples would be a date column, where it is known that anything older than a certain age isn't used, or a status flag column, where closed items are never queried. Currently we rely on special developer annotations as described in § 4.3. However, by detecting these columns and exploiting their semantics automatically we could automatically improve the performance without requiring any manual annotations.

Adaptive Range Filters [2] could be quite easily plugged into our system, and we would like to analyze their memory overhead vs. benefit to query performance.

The characteristics stored about aged data are quite minimal. We believe we can leverage much of the work done on materialized views in order to answer even more queries that need aggregated information about aged data without going to cold storage.

For the statistics, rather than store a single bit, a counter could be stored, indicating how many times an item was accessed. This would provide a better characterization of data access, but also increases the size of the statistics. We tested this approach, and in all the benchmarks the amount of data swelled to over five times the amount of simulated hot storage during a single aging period. It would be possible to have a counter with only a few discreet values (say [one, fewer than ten, ten or more]) which would not bloat the statistics size as much, and we plan to investigate this possibility.

9. CONCLUSION

We present Methuselah, a data aging system for in-memory databases. Our techniques impose minimal overhead on the running system, and fit well into the current data storage model used by these systems. Methuselah gathers attribute level access statistics without overly taxing the query runtime and uses compression to have minimal space requirements for their storage. Our experiments, using three different benchmarks, demonstrate our approach has excellent performance, keeping a high number of queries all in memory.

In short, we have presented an end-to-end system to manage both hot and cold data in an in-memory column store, covering the full pipeline from identifying cold data, moving it to cold storage, and then maintaining query performance in the face of cold data.

10. REFERENCES

[1] AuctionMark: An OLTP Benchmark for Shared-Nothing Database Management Systems. `http://hstore.cs.brown.edu/projects/auctionmark`. Accessed: 01/08/2012.

[2] Karolina Alexiou, Donald Kossmann, and Per-Ake Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *PVLDB*, 6(14):1714–1725, 2013.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[4] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 479–490, New York, NY, USA, 2006. ACM.

[5] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

[6] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[7] Asit Dan and Don Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 143–152, New York, NY, USA, 1990. ACM.

[8] Shaul Dar, Michael J. Franklin, Björn T. Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement, 1996.

[9] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-Caching: A new approach to database management system architecture. *Proc. VLDB Endow.*, 6:1942–1953, September 2013.

[10] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, New York, NY, USA, 2013. ACM.

[11] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[12] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core cpus. *CoRR*, abs/1109.6885, 2011.

[13] Justin J. Levandoski, Per-Ake Larson, and Radu Stoica. Identifying hot and cold data in main-memory databases. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering*, To appear in ICDE '13. IEEE Computer Society, 2013.

[14] Daniel A. Menasce. Tpc-w - a benchmark for e-commerce, 2002.

[15] Oracle. Working with Data in a TimesTen Database: Implementing aging in your tables. `http://docs.oracle.com/cd/E13085_01/doc/timesten.1121/e13065/comp.htm#CHDHIJBA`.

Accessed: 02/08/2012.

[16] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

[17] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 1–2, New York, NY, USA, 2009. ACM.

[18] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alexander Rasin, Nga Tran, and Stan B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, Trondheim, Norway, 2005.

[19] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The hp autoraid hierarchical storage system. In *ACM Transactions on Computer Systems*, pages 96–108, 1995.