Hewlett Packard Labs

Vertex: A Hybrid Database System for Mixed Workloads

Alkis Simitsis, Kevin Wilkinson

Hewlett Packard Labs
HPE-2016-91

Keyword(s):
Hybrid database; OLTP; OLAP; mixed workloads; workload management

Abstract:
Existing database management systems are designed for either transactional workloads or analytic workloads. They do not provide sustained, high performance for both workloads concurrently. We designed Vertex to achieve this. It uses a hybrid architecture that tightly couples an analytic engine with an OLTP engine and replicates tables across both engines. The analytic engine processes analytic queries. The OLTP engine processes OLTP transactions. OLTP table changes are replicated to analytic tables by a technique that enables high-speed bulk data insert. We describe Vertex and give performance results for a proof-of-concept that demonstrates concurrent, high performance on a mixed workload of TPC-C and TPC-H requests.

External Posting Date: October 25, 2016 [Fulltext]
Internal Posting Date: October 25, 2016 [Fulltext]

Hewlett Packard Enterprise

Vertex: A Hybrid Database System for Mixed Workloads

Alkis Simitsis, Kevin Wilkinson

Hewlett Packard Labs

September 20, 2016

Technical Report

Contents

Abstract

Existing database management systems are designed for either transactional workloads or analytic workloads. They do not provide sustained, high performance for both workloads concurrently. We designed Vertex to achieve this. It uses a hybrid architecture that tightly couples an analytic engine with an OLTP engine and replicates tables across both engines. The analytic engine processes analytic queries. The OLTP engine processes OLTP transactions. OLTP table changes are replicated to analytic tables by a technique that enables high-speed bulk data insert. We describe Vertex and give performance results for a proof-of-concept that demonstrates concurrent, high performance on a mixed workload of TPC-C and TPC-H requests.

Note that the information described in this document was last updated in *December, 2014*. It is possible that there have been several advances in technology and the software products used in this work since then.

Introduction

Analytic engines provide excellent performance for analytic, read-mostly, workloads. Typically, for other workloads these engines support a modest rate of update requests, as they were not designed for frequent updates. However, it is possible to extend analytic engines to support such workloads and presumably, enable new applications and increase their customer base. On the other hand, OLTP engines are designed for processing transactions at high throughput, but they are not optimized for heavy, analytic workloads.

Broadly speaking, there are two database system architectures for mixed workloads:

(1) One-engine-fits-all, where transactions and analytics are processed by the same engine

(2) A hybrid architecture with separate engines for each workload

The first approach is interesting, one that other vendors and research teams are currently investigating. However, it would require changes, possibly significant, to the database internals. The hybrid approach requires fewer, if any, engine changes. In addition, the use of different engines allows them to be tuned separately for each workload. However, it does require duplication and replication of data, which imposes a synchronization delay as updates are propagated.

Vertex is a hybrid database system for mixed workloads. It uses a high-speed bulk data insert method to replicate data and provides excellent performance on both workloads. Next, we describe a proof-of-concept implementation that uses HPE Vertica as an analytic engine and MySQL as an OLTP engine, and present performance results on a mixed workload of TPC-C and TPC-H requests.

Vertex

The Vertex architecture (see Figure 1) comprises two existing database engines plus additional modules for infrastructure. Our initial implementation uses Vertica to process analytic queries and MySQL for OLTP requests. We chose MySQL for quick prototyping; it can be replaced by another engine, e.g., a main-memory engine. Similarly, we could us an alternative engine for analytic queries although, as we explain later, Vertica has particular design features that work well in this architecture.

In Vertex, applications are not aware of the underlying engines. The Vertex API exposes a single system image. Applications send all requests to a Federation Layer that, in turns, redirects requests to the appropriate engine. Currently it acts as a simple workload manager and redirects requests based on a type tag (e.g., user or application identifier). For now, assume that each OLTP table is exactly replicated on the OLAP engine (extensions to general ETL are out of scope of this paper) with some time delay. A Vertex extract module collects committed OLTP row-level changes that occurred after the previous extract and creates a set of load files, formatted for the Vertica copy command. Another Vertex module

appends these files to Vertica (hot) tables as described below. This is done with very little interference on existing queries on MySQL and Vertica. A third module merges the new, hot, data with existing Vertica data and deletes older data that will no longer be queried (*garbage-collects*). The frequency of the extract, load and refresh operations can be varied independently.

We now provide more details on these operations. Consider a database table that is frequently updated. Looking closer, we observe that some attributes in the table are cold, i.e., static or rarely changed. Other attributes are hot, i.e., frequently updated. Looking further, we identify patterns in how the hot columns are modified.

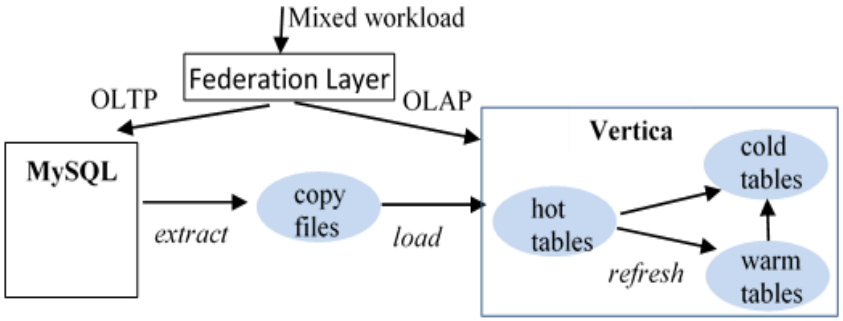

Figure 1. Vertex Architecture

Consider the TPC-C transactions, which are representative of OLTP workloads. We identified two data modification patterns that we refer to as *hot-spot* and *state-change*. A hot-spot models a long-lived business object in which one or more of its attributes frequently change while the other attributes remain stable (unchanged). A state-change models a business object that goes through a number of phases and then reaches a final state where it becomes stable. As before, some attributes remain stable while other attributes capture the state changes of the object.

An example hot-spot is the *ytd* attribute in the TPC-C Warehouse table (see Figure 2). It stores the warehouse year-to-date sales and is updated by every Payment transaction. The other warehouse attributes never change; we call them cold attributes. An example state-change is the *delivery-date* attribute in the Order_Line table. A New_Order transaction inserts multiple Order_Line rows, one per item ordered. However, the order is not stable until a Delivery transaction sets the *delivery-date* attribute. After that, it does not change. The other attributes of Order_Line are cold attributes because they do not change after insertion.

The analytic engine of Vertex stores each logical table as two physical tables, one with the cold attributes and a second with the hot attributes. A view is then provided so applications see one logical table. In this way, update operations only affect the hot table. We now describe how Vertex handles updates to the hot table and queries over the view. In Vertica, update and delete operations acquire an exclusive table lock. This is why updates are slow and this also limits throughput. However, insert operations are fast. No locks are required so inserts do not interfere with other concurrent operations. Therefore, we convert update and delete operations to inserts. This is done by adding two additional columns to the hot table, a timestamp and a delete flag. In effect, each row in the hot table is versioned.

Different techniques are used for hot-spot and state-change attributes. An OLTP table with hot-spots is decomposed into two OLAP tables, one with cold attributes, another with hot attributes. Both include the primary key attributes. The hot table includes a timestamp and a delete flag in its primary key. The Vertex extract process converts an OLTP hot attribute update to an OLAP hot table insert with a timestamp of the update. The freshest (newest) value of an OLAP hot attribute is that row with the most recent timestamp. In Vertica, this is efficiently done with a *Top-K Live Attribute Projection*. If the delete flag is set on that row, no value is returned. We do not discuss row delete in detail here. To provide the illusion of a single table, a view is created to join the hot and cold tables, which, in this case, can be an efficient merge join.

An OLTP table with state-change attributes is stored as three OLAP tables. The cold table contains attributes for stable (unchanging) objects (e.g., delivered orders). The hot table contains cold attributes for active objects, i.e., attributes that do not change after insert. A state table contains the state attributes, a timestamp and delete flag. All tables have the

primary key. Again, a view provides the illusion of a single table. Note that a state-change object is stored either in a cold table or in a hot table but not both. So, the cold and hot tables can be efficiently merged with a union operator. However, the hot table and its associated state change table must be joined. As an optimization, if a state change attribute is only updated once (e.g., *delivery-date*), no timestamp is needed for the state change table, and it may be joined directly with the hot table.

Over time, the hot and state tables will grow with old data. To address this, there is a refresh process. Each hot/state table has a corresponding warm table. Refresh truncates warm tables, copies the current hot/state values into the corresponding warm table and then truncates the hot/state tables. The warm table requires no timestamp column so it may be combined directly with its corresponding cold table. Warm tables improve performance for applications that require recent (warm) data but not necessarily the freshest (hot) data. Thus, applications can trade-off freshness for performance. It also enables a way to support ETL and/or application-level consistency constraints, e.g., surrogate keys, or applications that should only see paid orders, for example.

The refresh process can be optimized for state tables. For example, once an object reaches its terminal state, e.g., delivered, that object, in effect, becomes static or cold. So, those objects can be appended to a static table for improved performance. Other optimizations are possible. For example, there might be multiple state tables for a state object, one for each different possible state, e.g., ordered, paid-for, delivered. These tables would all be append-only and enable a quick determination of all objects in a particular state.

The Appendix describes in more detail an example application of this technique.

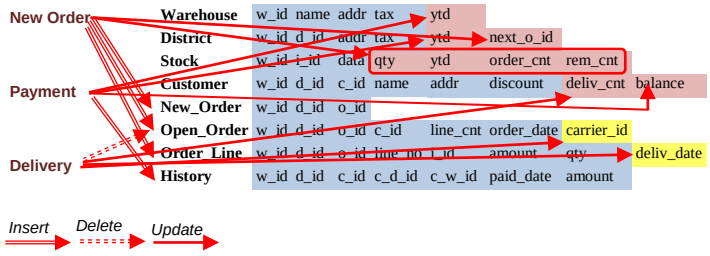

Figure 2. TPC-C Transaction Changes

Experiments

In this section, we present performance results for a proof-of-concept on a mixed workload of TPC-C and TPC-H requests.

Experimental Setup

A Vertex proof-of-concept (PoC) was evaluated on an 8-node Vertica cluster and a 1-node MySQL instance. Our mixed workload is CH-Benchmark [1], which combines TPC-C and a slightly modified TPC-H so that both workloads use a common schema. OLTP-Bench [2] is used to submit TPC-C requests to MySQL and TPC-H queries to Vertica. Periodically, a batch of row-level changes is extracted from the MySQL binary log and parsed to generate Vertica copy files. These files are then loaded into Vertica hot/state tables. TPC-H queries are directed either at the warm tables or at the freshest data in the hot tables. The TPC-C workload comprises five transactions: three modify data and two are queries. Figure 2 shows the cold, hot, and state attributes for the TPC-C tables, and also shows how the transactions affect them. All state attributes change only once so the optimization mentioned above applies.

The PoC addresses two main questions:

1) First, can the extract and load processes keep pace with the OLTP transaction rate. If Vertica falls behind, it will be impossible to provide freshness guarantees.

2) Second, compared to querying base tables, what is the performance penalty for querying warm data or the fresh data. If the overhead of the additional joins is too large, Vertex is not a viable solution.

Freshness

To address the first question, an experiment measured the freshness of data loaded into Vertica while varying the transaction rate and the frequency of extract and load. Freshness is defined as the delay in seconds between the commit of a History table row on MySQL and its appearance in the hot History table in Vertica.

Each second, a Vertica query looked for new hot History rows and reported the difference between the current time and the MySQL commit time for that row. In the first run, MySQL processed 150 transaction per second (about 4000 TPC-C tpm) and the extract-and-load frequency was fixed at 5 seconds. In a second run, the frequency was 10 seconds. In a third run, a heavier load, MySQL processed 1000 transactions per second (21,000 tpm) and the frequency was 5 seconds.

If Vertex is keeping pace, we expect the freshness to be a few seconds slower than the extract frequency (a few seconds are needed for extract and load) and to be remain at steady state. Figure 3 shows that freshness is close to the extract frequency for 150 tps. For a heavy load of 900 tps, freshness is also close to the extract frequency. An investigation showed that the extract and copy always completed within a second or two, but parsing the extract to generate the copy files was slower. So for very heavy transaction rates, this could be limiting. It may have to be addressed for a future Vertex using a main-memory OLTP engine with, presumably, higher transaction rates. However, it is a simple matter to implement faster parsing, so we feel confident that Vertex can keep pace for heavier loads.

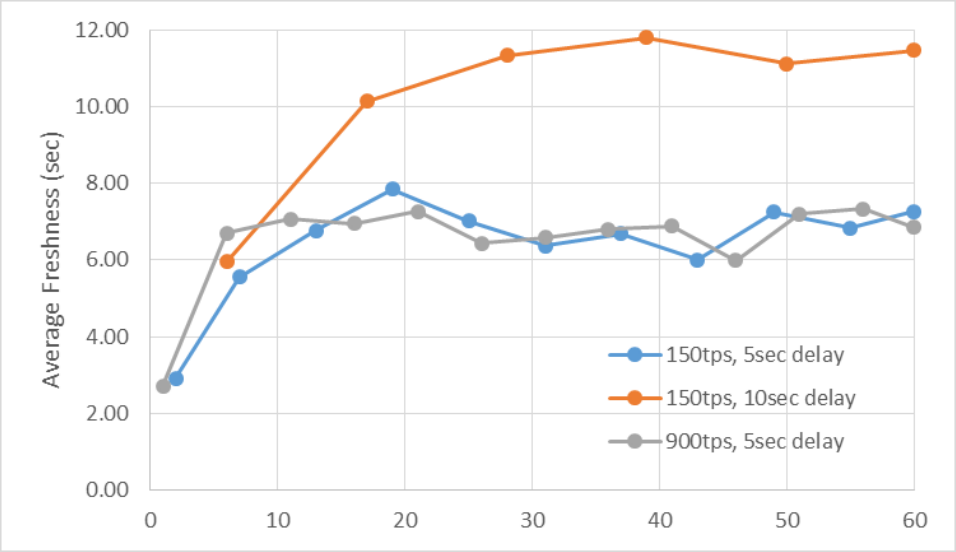

Figure 3. Freshness during 1 minute run

Query Latency

To address the second question, we measured TPC-H query latency on Vertica on warm data and hot data and compared this to a baseline of TPC-H on a conventional schema (see Figure 4). The Vertica Database Designer was used for each query set. The (TPC-C) scale factor was 200 and hot data was loaded for 45,000 TPC-C transactions (5 minutes at 150tps).

On average, the warm queries were two times slower than baseline (or 1.7 slower discounting the outlier, Q17). The hot queries were almost 2.5 times slower than baseline and 1.3 times slower than warm. Investigation showed that large time differences were due to different query plans rather than the overhead of joining cold, warm and hot tables. In particular, predicate evaluation was not pushed down to the table scan for some warm and hot queries so entire tables had to be read and joined. This can be fixed with query-specific tuning but we are working with Vertica to improve their optimizer to

handle such queries. More work is needed but we think the results are encouraging. Note that since Vertex essentially mimics what Vertica already does with its containers and delete vectors, we think Vertex could be implemented inside Vertica for even better performance.

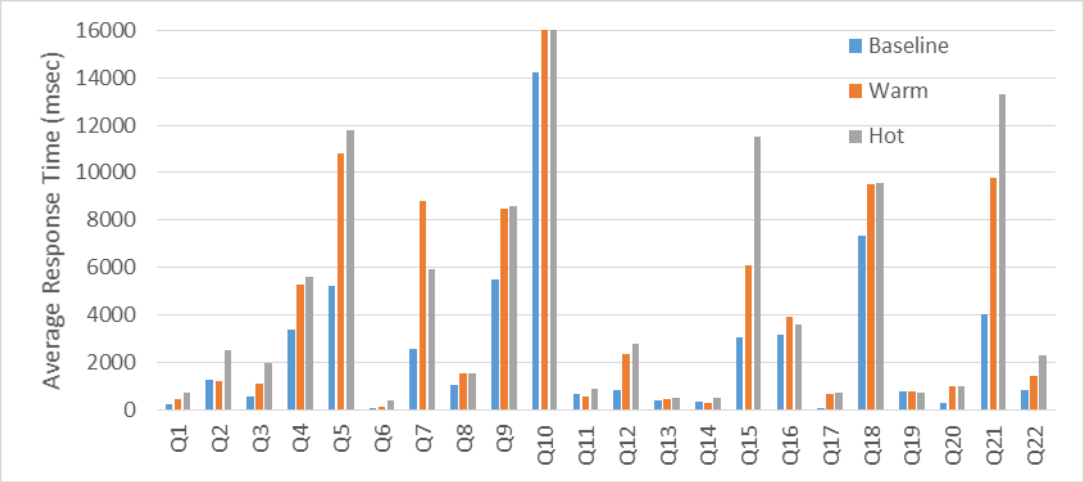

Figure 4. Baseline vs. Warm and Hot on (Modified) TPC-H

Related Work

Systems to support mixed workloads have been proposed and implemented. The HyPer system [3] stores all data in main memory and it uses versioning to ensure that applications view consistent data. However, it uses the operating system to create copies of data pages when updates occur so this versioning is coarse-grained since data pages are on the order of thousands of bytes. SAP HANA is another main-memory system for mixed workloads [4]. In this system, two versions of the data are maintained, one for high frequency update operations and a second for analysis and reports operations. The Vertex advantage is that it leverages commodity hardware and the OLAP database size is not contained by the size of main memory. Tungsten [5] provides real-time MySQL-to-Vertica replication that supports update and delete but applied directly to Vertica base tables.

Vertica Database Designer is an automated tool to design table partitions, but, while it may consider updates, does not distinguish hot and cold attributes. Internally, Vertica implements delete operations by inserting a delete flag to indicate a row is invalid. Updates are implemented as a delete followed by an insert of the new values. Therefore, like our approach, Vertica stores multiple versions of data and uses delete flags to denote old versions. Therefore, in a sense, our approach is an external adaptation of what Vertica does internally. However, a Vertica delete operation requires an exclusive table lock, which delays other, concurrent requests and so greatly reduces overall performance. Similarly, Vertica update operations degrade system performance because they are implemented as the composition of a delete operation and an insert operation. Our update technique avoids the expensive delete operations.

Conclusions

A general-purpose data management system must support four primitive operations: read (a set of records), insert, update, and delete. Databases used for business reporting and analysis are typically read-optimized data stores. Large enterprises have many such databases. A read-optimized system provides excellent performance for operations that read a large amount of data. Typically, it also provides good performance for insertion of data. However, delete and update operations are typically slow and they interfere with read and insert performance. To reduce contention with read operations, updates are typically deferred to periods of low activity (e.g., nights or weekends).

Vertex is a hybrid database system for mixed, analytical and transactional workloads. It tightly couples an analytic engine with an OLTP engine and replicates tables across both engines. Vertex addresses the problem of updating a “read-optimized” database engine with a technique that converts delete and update operations to efficient insert operations, thus

improving overall system performance. Vertex also enables more frequent updates, which makes new data available sooner for reporting and analysis. Our PoC shows that Vertex is a feasible approach.

Interesting future directions include support for additional data modification patterns and automation to generate the extract, load, and refresh scripts.

References

[1] Dagstuhl Breakout Group. The mixed workload CH-Benchmark. In DBTest Workshop, Athens, 2011.

[2] D.E. Difallah et al. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. In VLDB, 2013.

[3] A. Kemper, T. Neumann. HyPer: A Hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In IEEE ICDE, 2011.

[4] V. Sikka et al. Efficient Transaction processing in SAP HANA database. In ACM SIGMOD, 2012.

[5] Continuent Tungsten product. http://scale-out-blog.blogspot.com/2012/06/mysql-to-vertica-replication-part-1.html

Appendix: An Example Application

We illustrate Vertex technique for updating a read-optimized data store with an example. Consider the “Stock” table from the TPC-C benchmark. It records the stock level (i.e., number available) of each item in a warehouse. It comprises the following attributes:

```
S_W_ID INT                      warehouse identifier
S_I_ID INT                      item identifier
S_QUANTITY DECIMAL(4,0)         quantity available
S_YTD DECIMAL(8,2)              quantity sold year-to-date
S_ORDER_CNT INT                 number of orders for this item
S_REMOTE_CNT INT                number of orders for this item from a different warehouse
S_DATA VARCHAR(50)              item description
S_DIST_01 CHAR(24)              item-specific data for warehouse district 1
S_DIST_02 CHAR(24)              item-specific data for warehouse district 2
S_DIST_03 CHAR(24)              item-specific data for warehouse district 3
S_DIST_04 CHAR(24)              item-specific data for warehouse district 4
S_DIST_05 CHAR(24)              item-specific data for warehouse district 5
S_DIST_06 CHAR(24)              item-specific data for warehouse district 6
S_DIST_07 CHAR(24)              item-specific data for warehouse district 7
S_DIST_08 CHAR(24)              item-specific data for warehouse district 8
S_DIST_09 CHAR(24)              item-specific data for warehouse district 9
S_DIST_10 CHAR(24)              item-specific data for warehouse district 10
PRIMARY KEY (S_W_ID,S_I_ID)
```

For each item in a purchase order, the item record in the supplying warehouse is updated. Specifically, the quantity, year-to-date count, order count and remote counts are updated. Therefore, these attributes constitute the hot data for the STOCK table. Consequently, we could replace the STOCK table with the following schema.

```
CREATE TABLE STOCK_HOT (
  S_W_ID INT
  S_I_ID INT
  S_QUANTITY DECIMAL(4,0)
  S_YTD DECIMAL(8,2)
  S_ORDER_CNT INT
  S_REMOTE_CNT INT
```

```
  S_TS TIMESTAMP
  S_DF BOOLEAN
  PRIMARY KEY (S_W_ID,S_I_ID));

CREATE TABLE STOCK_COLD (
  S_W_ID INT
  S_I_ID INT
  S_DATA VARCHAR(50)
  S_DIST_01 CHAR(24)
  S_DIST_02 CHAR(24)
  S_DIST_03 CHAR(24)
  S_DIST_04 CHAR(24)
  S_DIST_05 CHAR(24)
  S_DIST_06 CHAR(24)
  S_DIST_07 CHAR(24)
  S_DIST_08 CHAR(24)
  S_DIST_09 CHAR(24)
  S_DIST_10 CHAR(24)
  PRIMARY KEY (S_W_ID,S_I_ID));

CREATE VIEW STOCK AS
      SELECT C.S_W_ID, C.S_I_ID,
            H.S_QUANTITY, H.S_YTD, H.S_ORDER_CNT, H.S_REMOTE_CNT, C.S_DATA,
            C.S_DIST_01, C.S_DIST_02, C.S_DIST_03, C.S_DIST_04, C.S_DIST_05,
            C.S_DIST_06, C.S_DIST_07, C.S_DIST_08, C.S_DIST_09, C.S_DIST_10
      FROM STOCK_CLD C,
            (SELECT H.S_W_ID, H.S_I_ID, H.S_QUANTITY, H.S_YTD,
                  H.S_ORDER_CNT, H.S_REMOTE_CNT, max(H.S_TS) as S_TS, S_DF
            FROM STOCK_HOT H
            GROUP BY H.S_W_ID, H.S_I_ID, H.S_QUANTITY, H.S_YTD,
                  H.S_ORDER_CNT, H.S_REMOTE_CNT) H
      WHERE H.S_I_ID = C.S_I_ID AND H.S_W_ID = C.S_W_ID AND S_DF = FALSE;
```

The view is used by analytic queries to retrieve all attributes of the current stock data. It retrieves the latest version of the hot attributes for each stock item in a warehouse (max(H.S_TS)). It then merges these with the corresponding cold attributes by joining on the primary key. The delete flag, S_DF, is discussed below.

When a stock item is updated, the updated values are inserted as a new row in the STOCK_HOT table with the timestamp of the update and the delete flag set to false. When a stock item is deleted, a new row is inserted into the STOCK_HOT table with the timestamp of the deletion and with the delete flag set to true. The delete flag indicates that the row is deleted and should not be returned by the STOCK view.

Over time, the STOCK_HOT table will grow as it accumulates old versions of attributes. Eventually, the old versions will have to be deleted (garbage collected). This is easily accomplished by simply copying the hot table, ignoring those rows where the delete flag is true. Further, this can be done concurrently with other query requests. It only requires a short, atomic schema operation to rename STOCK_HOT to STOCK_HOT_OLD and the creation of a new STOCK_HOT table. At this point, updates may resume on the (empty) STOCK_HOT table. Then, the STOCK_HOT_OLD table can be copied to STOCK_HOT, in the process dropping old attribute values. Note that during this copy, a different variant of the STOCK view must be used, i.e., one that reads from both STOCK_HOT_OLD and STOCK_HOT (this modified view is not shown). Once the copy is complete, a second, atomic schema operation is needed to drop STOCK_HOT_OLD and restore the original view definition for STOCK.

Given this basic framework, numerous optimizations are possible depending on the needs of the application. For example, since the hot table records old versions of attribute values, applications may choose to query the stock level at some previous point in time. Alternatively, some applications may prefer faster query performance rather than have the

freshest data. In this case, it is possible to keep a second copy of the Stock table that has both hot and cold attributes but with older data. Applications can read stock level data directly from this table without the overhead of a group by and join operation. This table can be periodically updated using a technique like that for garbage collection. Another optimization concerns updates to cold attributes. Recall that cold attributes are updated using the default technique for the data store. However, it would be possible to include a timestamp and delete flag on cold attributes as well. This would improve update performance at the expense of read performance (since reading the cold attributes would now also require a group-by operation to get the latest value). However, for some applications, this may provide better overall performance.

For read requests, our technique is transparent to applications. However, update requests may need rewriting to reference new target tables. For example, a simple insert of a single row to the Stock table now becomes two insert operations to the cold and hot tables. A Stock delete operation becomes a delete on the cold table and an insert on the hot table. An update on hot attributes becomes an insert on the hot table and so on. This rewriting could be done by middleware.