The End of an Architectural Era (It's Time for a Complete Rewrite)

Michael Stonebraker
Samuel Madden
Daniel J. Abadi
Stavros Harizopoulos
MIT CSAIL
{stonebraker, madden, dna, stavros}@csail.mit.edu

Nabil Hachem
AvantGarde Consulting, LLC
nhachem@agdba.com

Pat Helland
Microsoft Corporation
phelland@microsoft.com

ABSTRACT

In previous papers [SC05, SBC+07], some of us predicted the end of "one size fits all" as a commercial relational DBMS paradigm. These papers presented reasons and experimental evidence that showed that the major RDBMS vendors can be outperformed by 1-2 orders of magnitude by specialized engines in the data warehouse, stream processing, text, and scientific database markets.

Assuming that specialized engines dominate these markets over time, the current relational DBMS code lines will be left with the business data processing (OLTP) market and hybrid markets where more than one kind of capability is required. In this paper we show that current RDBMSs can be beaten by nearly two orders of magnitude in the OLTP market as well. The experimental evidence comes from comparing a new OLTP prototype, H-Store, which we have built at M.I.T. to a popular RDBMS on the standard transactional benchmark, TPC-C.

We conclude that the current RDBMS code lines, while attempting to be a "one size fits all" solution, in fact, excel at nothing. Hence, they are 25 year old legacy code lines that should be retired in favor of a collection of "from scratch" specialized engines. The DBMS vendors (and the research community) should start with a clean sheet of paper and design systems for tomorrow's requirements, not continue to push code lines and architectures designed for yesterday's needs.

1. INTRODUCTION

The popular relational DBMSs all trace their roots to System R from the 1970s. For example, DB2 is a direct descendent of System R, having used the RDS portion of System R intact in their first release. Similarly, SQL Server is a direct descendent of Sybase System 5, which, borrowed heavily from System R. Lastly, the first release of Oracle implemented the user interface from System R.

All three systems were architected more than 25 years ago, when hardware characteristics were much different than today. Processors are thousands of times faster and memories are thousands of times larger. Disk volumes have increased enormously, making it possible to keep essentially everything, if one chooses to. However, the bandwidth between disk and main memory has increased much more slowly. One would expect this relentless pace of technology to have changed the architecture of database systems dramatically over the last quarter of a century, but surprisingly the architecture of most DBMSs is essentially identical to that of System R.

Moreover, at the time relational DBMSs were conceived, there was only a single DBMS market, business data processing. In the last 25 years, a number of other markets have evolved, including data warehouses, text management, and stream processing. These markets have very different requirements than business data processing.

Lastly, the main user interface device at the time RDBMSs were architected was the dumb terminal, and vendors imagined operators inputting queries through an interactive terminal prompt. Now it is a powerful personal computer connected to the World Wide Web. Web sites that use OLTP DBMSs rarely run interactive transactions or present users with direct SQL interfaces.

In summary, the current RDBMSs were architected for the business data processing market in a time of different user interfaces and different hardware characteristics. Hence, they all include the following System R architectural features:

- Disk oriented storage and indexing structures
- Multithreading to hide latency
- Locking-based concurrency control mechanisms
- Log-based recovery

Of course, there have been some extensions over the years, including support for compression, shared-disk architectures, bitmap indexes, support for user-defined data types and operators, etc. However, no system has had a complete redesign since its inception. This paper argues that the time has come for a complete rewrite.

A previous paper [SBC+07] presented benchmarking evidence that the major RDBMSs could be beaten by specialized

VLDB '07, September 23-28, 2007, Vienna, Austria.

architectures by an order of magnitude or more in several application areas, including:

- Text (specialized engines from Google, Yahoo, etc.)
- Data Warehouses (column stores such as Vertica, Monet [Bon02], etc.)
- Stream Processing (stream processing engines such as StreamBase and Coral8)
- Scientific and intelligence databases (array storage engines such as MATLAB and ASAP [SBC+07])

Based on this evidence, one is led to the following conclusions:

1) RDBMSs were designed for the business data processing market, which is their sweet spot
2) They can be beaten handily in most any other market of significant enough size to warrant the investment in a specialized engine

This paper builds on [SBC+07] by presenting evidence that the current architecture of RDBMSs is not even appropriate for business data processing. Our methodology is similar to the one employed in [SBC+07]. Specifically, we have designed a new DBMS engine for OLTP applications. Enough of this engine, H-Store, is running to enable us to conduct a performance bakeoff between it and a popular commercial RDBMSs. Our experimental data shows H-Store to be a factor of 82 faster on TPC-C (almost two orders of magnitude).

Because RDBMSs can be beaten by more than an order of magnitude on the standard OLTP benchmark, then there is no market where they are competitive. As such, they should be considered as legacy technology more than a quarter of a century in age, for which a complete redesign and re-architecting is the appropriate next step.

Section 2 of this paper explains the design considerations that can be exploited to achieve this factor of 82 on TPC-C. Then, in Section 3, we present specific application characteristics which can be leveraged by a specialized engine. Following that, we sketch some of the H-store design in Section 4. We then proceed in Section 5 to present experimental data on H-Store and a popular RDBMS on TPC-C. We conclude the paper in Section 6 with some radical suggestions for the research agenda for the DBMS community.

2. OLTP Design Considerations

This section presents five major issues, which a new engine such as H-Store can leverage to achieve dramatically better performance than current RDBMSs.

2.1 Main Memory

In the late 1970's a large machine had somewhere around a megabyte of main memory. Today, several Gbytes are common and large machines are approaching 100 Gbytes. In a few years a terabyte of main memory will not be unusual. Imagine a shared nothing grid system of 20 nodes, each with 32 Gbytes of main memory now, (soon to be 100 Gbytes), and costing less than $50,000. As such, any database less than a terabyte in size, is capable of main memory deployment now or in the near future.

The overwhelming majority of OLTP databases are less than 1 Tbyte in size and growing in size quite slowly. For example, it is a telling statement that TPC-C requires about 100 Mbytes per physical distribution center (warehouse). A very large retail enterprise might have 1000 warehouses, requiring around 100 Gbytes of storage, which fits our envelope for main memory deployment.

As such, we believe that OLTP should be considered a main memory market, if not now then within a very small number of years. Consequently, the current RDBMS vendors have disk-oriented solutions for a main memory problem. In summary, 30 years of Moore's law has antiquated the disk-oriented relational architecture for OLTP applications.

Although there are some main memory database products on the market, such as TimesTen and SolidDB, these systems inherit the baggage of System R as well. This includes such features as a disk-based recovery log and dynamic locking, which, as we discuss in the following sections, impose substantial performance overheads.

2.2 Multi-threading and Resource Control

OLTP transactions are very lightweight. For example, the heaviest transaction in TPC-C reads about 200 records. In a main memory environment, the useful work of such a transaction consumes less than one millisecond on a low-end machine. In addition, most OLTP environments we are familiar with do not have "user stalls". For example, when an Amazon user clicks "buy it", he activates an OLTP transaction which will only report back to the user when it finishes. Because of an absence of disk operations and user stalls, the elapsed time of an OLTP transaction is minimal. In such a world it makes sense to run each SQL command in a transaction to completion with a single-threaded execution model, rather than paying for the overheads of isolation between concurrently executing statements.

Current RDBMSs have elaborate multi-threading systems to try to fully utilize CPU and disk resources. This allows several-to-many queries to be running in parallel. Moreover, they also have resource governors to limit the multiprogramming load, so that other resources (IP connections, file handles, main memory for sorting, etc.) do not become exhausted. These features are irrelevant in a single threaded execution model. No resource governor is required in a single threaded system.

In a single-threaded execution model, there is also no reason to have multi-threaded data structures. Hence the elaborate code required to support, for example, concurrent B-trees can be completely removed. This results in a more reliable system, and one with higher performance.

At this point, one might ask "What about long running commands?" In real-world OLTP systems, there aren't any for two reasons: First, operations that appear to involve long-running transactions, such as a user inputting data for a purchase on a web store, are usually split into several transactions to keep transaction time short. In other words, good application design will keep OLTP queries small. Second, longer-running ad-hoc queries are not processed by the OLTP system; instead such queries are directed to a data warehouse system, optimized for this activity. There is no reason for an OLTP system to solve a non-OLTP problem. Such thinking only applies in a "one size fits all" world.

2.3 Grid Computing and Fork-lift Upgrades

Current RDBMSs were originally written for the prevalent architecture of the 1970s, namely shared-memory multiprocessors. In the 1980's shared disk architectures were spearheaded by Sun and HP, and most DBMSs were expanded to include capabilities for this architecture. It is obvious that the

next decade will bring domination by shared-nothing computer systems, often called grid computing or blade computing. Hence, any DBMS must be optimized for this configuration. An obvious strategy is to horizontally partition data over the nodes of a grid, a tactic first investigated in Gamma [DGS+90].

In addition, no user wants to perform a "fork-lift" upgrade. Hence, any new system should be architected for incremental expansion. If N grid nodes do not provide enough horsepower, then one should be able to add another K nodes, producing a system with N+K nodes. Moreover, one should perform this upgrade, without a hiccup, i.e. without taking the DBMS down. This will eliminate every system administrator's worst nightmare; a fork-lift upgrade with a requirement for a complete data reload and cutover.

To achieve incremental upgrade without going down requires significant capabilities, not found in existing systems. For example, one must be able to copy portions of a database from one site to another without stopping transactions. It is not clear how to bolt such a capability onto most existing systems. However, this can be made a requirement of a new design and implemented efficiently, as has been demonstrated by the existence of exactly this feature in the Vertica[1] codeline.

2.4 High Availability

Relational DBMSs were designed in an era (1970s) when an organization had a single machine. If it went down, then the company lost money due to system unavailability. To deal with disasters, organizations typically sent log tapes off site. If a disaster occurred, then the hardware vendor (typically IBM) would perform heroics to get new hardware delivered and operational in small numbers of days. Running the log tapes then brought the system back to something approaching where it was when the disaster happened.

A decade later in the 1980's, organizations executed contracts with disaster recovery services, such as Comdisco, for backup machine resources, so the log tapes could be installed quickly on remote backup hardware. This strategy minimized the time that an enterprise was down as a result of a disaster.

Today, there are numerous organizations that run a **hot standby** within the enterprise, so that real-time failover can be accomplished. Alternately, some companies run multiple primary sites, so failover is even quicker. The point to be made is that businesses are much more willing to pay for multiple systems in order to avoid the crushing financial consequences of down time, often estimated at thousands of dollars per minute.

In the future, we see high availability and built-in disaster recovery as essential features in the OLTP (and other) markets. There are a few obvious conclusions to be drawn from this statement. First, every OLTP DBMS will need to keep multiple replicas consistent, requiring the ability to run seamlessly on a grid of geographically dispersed systems.

Second, most existing RDBMS vendors have glued multi-machine support onto the top of their original SMP architectures. In contrast, it is clearly more efficient to start with shared-nothing support at the bottom of the system.

Third, the best way to support shared nothing is to use multiple machines in a peer-to-peer configuration. In this way, the OLTP load can be dispersed across multiple machines, and inter-machine replication can be utilized for fault tolerance. That way, all machine resources are available during normal operation. Failures only cause degraded operation with fewer resources. In contrast, many commercial systems implement a "hot standby", whereby a second machine sits effectively idle waiting to take over if the first one fails. In this case, normal operation has only half of the resources available, an obviously worse solution. These points argue for a complete redesign of RDBMS engines so they can implement peer-to-peer HA in the guts of a new architecture.

In an HA system, regardless of whether it is hot-standby or peer-to-peer, logging can be dramatically simplified. One must continue to have an undo log, in case a transaction fails and needs to roll back. However, the undo log does not have to persist beyond the completion of the transaction. As such, it can be a main memory data structure that is discarded on transaction commit. There is never a need for redo, because that will be accomplished via network recovery from a remote site. When the dead site resumes activity, it can be refreshed from the data on an operational site.

A recent paper [LM06] argues that failover/rebuild is as efficient as redo log processing. Hence, there is essentially no downside to operating in this manner. In an HA world, one is led to having no persistent redo log, just a transient undo one. This dramatically simplifies recovery logic. It moves from an Aries-style [MHL+92] logging system to new functionality to bring failed sites up to date from operational sites when they resume operation.

Again, a large amount of complex code has been made obsolete, and a different capability is required.

2.5 No Knobs

Current systems were built in an era where resources were incredibly expensive, and every computing system was watched over by a collection of wizards in white lab coats, responsible for the care, feeding, tuning and optimization of the system. In that era, computers were expensive and people were cheap. Today we have the reverse. Personnel costs are the dominant expense in an IT shop.

As such "self-everything" (self-healing, self-maintaining, self-tuning, etc.) systems are the only answer. However, all RDBMSs have a vast array of complex tuning knobs, which are legacy features from a bygone era. True; all vendors are trying to provide automatic facilities which will set these knobs without human intervention. However, legacy code cannot ever remove features. Hence, "no knobs" operation will be in addition to "human knobs" operation, and result in even more system documentation. Moreover, at the current time, the automatic tuning aids in the RDBMSs that we are familiar with do not produce systems with anywhere near the performance that a skilled DBA can produce. Until the tuning aids get vastly better in current systems, DBAs will turn the knobs.

A much better answer is to completely rethink the tuning process and produce a new system with no visible knobs.

[1] http://www.vertica.com

3. Transaction, Processing and Environment Assumptions

If one assumes a grid of systems with main memory storage, built-in high availability, no user stalls, and useful transaction work under 1 millisecond, then the following conclusions become evident:

1) A persistent redo log is almost guaranteed to be a significant performance bottleneck. Even with group commit, forced writes of commit records can add milliseconds to the runtime of each transaction. The HA/failover system discussed earlier dispenses with this expensive architectural feature.
2) With redo gone, getting transactions into and out of the system is likely to be the next significant bottleneck. The overhead of JDBC/ODBC style interfaces will be onerous, and something more efficient should be used. In particular, we advocate running application logic – in the form of stored procedures – "in process" inside the database system, rather than the inter-process overheads implied by the traditional database client / server model.
3) An undo log should be eliminated wherever practical, since it will also be a significant bottleneck.
4) Every effort should be made to eliminate the cost of traditional dynamic locking for concurrency control, which will also be a bottleneck.
5) The latching associated with multi-threaded data structures is likely to be onerous. Given the short runtime of transactions, moving to a single threaded execution model will eliminate this overhead at little loss in performance.
6) One should avoid a two-phase commit protocol for distributed transactions, wherever possible, as network latencies imposed by round trip communications in 2PC often take on the order of milliseconds.

Our ability to remove concurrency control, commit processing and undo logging depends on several characteristics of OLTP schemas and transaction workloads, a topic to which we now turn.

3.1 Transaction and Schema Characteristics

H-Store requires the complete workload to be specified in advance, consisting of a collection of transaction *classes***.** Each class contains transactions with the same SQL statements and program logic, differing in the run-time constants used by individual transactions. Since there are assumed to be no ad-hoc transactions in an OLTP system, this does not appear to be an unreasonable requirement. Such transaction classes must be registered with H-Store in advance, and will be disallowed if they contain user stalls (transactions may contain stalls for other reasons – for example, in a distributed setting where one machine must wait for another to process a request.) Similarly, H-Store also assumes that the collection of tables (logical schema) over which the transactions operate is known in advance.

We have observed that in many OLTP workloads every table except a single one called the *root*, has exactly one join term which is a 1-n relationship to its ancestor. Hence, the schema is a *tree* of 1-n relationships. We denote this class of schemas as *tree schemas.* Such schemas are popular; for example, customers produce orders, which have line items and fulfillment schedules. Tree schemas have an obvious horizontal partitioning over the nodes in a grid. Specifically, the root table can be range or hash partitioned on the primary key(s). Every descendent table can be partitioned such that all equi-joins in the tree span only a single site. In the discussion to follow, we will consider both tree and non-tree schemas.

In a tree schema, suppose every command in every transaction class has equality predicates on the primary key(s) of the root node (for example, in an e-commerce application, many commands will be rooted with a specific customer, so will include predicates like `customer_id = 27`). Using the horizontal partitioning discussed above, it is clear that in this case every SQL command in every transaction is local to one site. If, in addition, every command in each transaction class is limited to the same single site, then we call the application a *constrained tree application (CTA).* A CTA application has the valuable feature that every transaction can be run to completion at a single site. The value of such *single-sited* transactions, as will be discussed in Section 4.3, is that transactions can execute without any stalls for communication with another grid site (however, in some cases, replicas will have to synchronize so that transactions are executed in the same order).

If every command in every transaction of a CTA specifies an equality match on the primary key(s) of one or more direct descendent nodes in addition to the equality predicate on the root, then the partitioning of a tree schema can be extended hierarchically to include these direct descendent nodes. In this case, a finer granularity partitioning can be used, if desired.

CTAs are an important class of single-sited applications which can be executed very efficiently. Our experience with many years of designing database applications in major corporations suggests that OLTP applications are often designed explicitly to be CTAs, or that decompositions to CTAs are often possible [Hel07]. Besides simply arguing that CTAs are prevalent, we are also interested in techniques that can be used to make non-CTA applications single-sited; it is an interesting research problem to precisely characterize the situations in which this is possible. We mention two possible schema transformations that can be systematically applied here.

First, consider all of the read-only tables in the schema, i.e. ones which are not updated by any transaction class. These tables can be replicated at all sites. If the application becomes CTA with these tables removed from consideration, then the application becomes single-sited after replication of the read-only tables.

Another important class of applications are *one-shot*. These applications have the property that all of their transactions can be executed in parallel without requiring intermediate results to be communicated among sites. Moreover, the result of previous SQL queries are never required in subsequent commands. In this case, each transaction can be decomposed into a collection of single-site plans which can be dispatched to the appropriate sites for execution.

Applications can often be made one-shot with vertical partitioning of tables amongst sites (columns that are not updated are replicated); this is true of TPC-C, for example (as we discuss in Section 5.)

Some transaction classes are *two-phase* (or can be made to be two phase.) In phase one there are a collection of read-only operations. Based on the result of these queries, the transaction may be aborted. Phase two then consists of a collection of queries and updates where there can be no possibility of an integrity violation. H-Store will exploit the two-phase property to

eliminate the undo log. We have observed that many transactions, including those in TPC-C, are two-phase.

A transaction class is *strongly two-phase* if it is two-phase and additionally has the property that phase 1 operations on all replicas result in all replica sites aborting or all continuing.

Additionally, for every transaction class, we find all other classes whose members *commute* with members of the indicated class. Our specific definition of commutativity is:

> Two concurrent transactions from the same or different classes *commute* when any interleaving of their single-site sub-plans produces the same final database state as any other interleaving (assuming both transactions commit).

A transaction class which commutes with all transaction classes (including itself) will be termed *sterile*.

We use single-sited, sterile, two-phase, and strong two-phase properties in the H-Store algorithms, which follow. We have identified these properties as being particularly relevant based on our experience with major commercial online retail applications, and are confident that they will be found in many real world environments.

4. H-Store Sketch

In this section, we describe how H-Store exploits the previously described properties to implement a very efficient OLTP database.

4.1 System Architecture

H-Store runs on a grid of computers. All objects are partitioned over the nodes of the grid. Like C-Store [SAB+05], the user can specify the level of K-safety that he wishes to have.

At each site in the grid, rows of tables are placed contiguously in main memory, with conventional B-tree indexing. B-tree block size is tuned to the width of an L2 cache line on the machine being used. Although conventional B-trees can be beaten by cache conscious variations [RR99, RR00], we feel that this is an optimization to be performed only if indexing code ends up being a significant performance bottleneck.

Every H-Store site is single threaded, and performs incoming SQL commands to completion, without interruption. Each site is decomposed into a number of logical sites, one for each available core. Each logical site is considered an independent physical site, with its own indexes and tuple storage. Main memory on the physical site is partitioned among the logical sites. In this way, every logical site has a dedicated CPU and is single threaded.

In an OLTP environment most applications use stored procedures to cut down on the number of round trips between an application and the DBMS. Hence, H-Store has only one DBMS capability, namely to execute a predefined transaction (transactions may be issued from any site):

```
Execute transaction (parameter_list)
```

In the current prototype, stored procedures are written in C++, though we have suggestions on better languages in Section 6. Our implementation mixes application logic with direct manipulation of the database in the same process; this provides comparable performance to running the whole application inside a single stored procedure, where SQL calls are made as local procedure calls (not JDBC) and data is returned in a shared data array (again not JDBC).

Like C-Store there is no redo log, and an undo log is written only if required, as discussed in Section 4.4. If written, the undo log is main memory resident, and discarded on transaction commit.

4.2 Query Execution

We expect to build a conventional cost-based query optimizer which produces query plans for the SQL commands in transaction classes at transaction definition time. We believe that this optimizer can be rather simple, as 6 way joins are never done in OLTP environments. If multi-way joins occur, they invariably identify a unique tuple of interest (say a purchase order number) and then the tuples that join to this record (such as the line items). Hence, invariably one proceeds from an *anchor tuple* through a small number of 1-to-*n* joins to the tuples of ultimate interest. `GROUP BY` and aggregation rarely occur in OLTP environments. The net result is, of course, a simple query execution plan.

The query execution plans for all commands in a transaction may be:

> *Single-sited*: In this case the collection of plans can be dispatched to the appropriate site for execution.
>
> *One shot*: In this case, all transactions can be decomposed into a set of plans that are executed only at a single site.
>
> *General*: In the general case, there will be commands which require intermediate results to be communicated among sites in the grid. In addition, there may be commands whose run-time parameters are obtained from previous commands. In this case, we need the standard Gamma-style run time model of an *execution supervisor* at the site where the transaction enters the system, communicating with *workers* at the sites where data resides.

For general transactions, we compute the *depth* of the transaction class to be the number of times in the collection of plans, where a message must be sent between sites.

4.3 Database Designer

To achieve no-knobs operation, H-Store will build an automatic physical database designer which will specify horizontal partitioning, replication locations, and indexed fields.

In contrast to C-Store which assumed a world of overlapping materialized views appropriate in a read-mostly environment, H-Store implements the tables specified by the user and uses standard replication of user-specified tables to achieve HA. Most tables will be horizontally partitioned across all of the nodes in a grid. To achieve HA, such *table fragments* must have one or more *buddies*, which contain exactly the same information, possibly stored using a different physical representation (e.g., sort order).

The goal of the database designer is to make as many transaction classes as possible single-sited. The strategy to be employed is similar to the one used by C-Store [SAB+05]. That system constructed automatic designs for the omnipresent star or snowflake schemas in warehouse environments, and is now in the process of generalizing these algorithms for schemas that are "near snowflakes". Similarly, H-Store will construct automatic designs for the common case in OLTP environments (constrained tree applications), and will use the previously mentioned strategy of partitioning the database across sites based on the primary key of the root table and assigning tuples of other tables to sites based on root tuples they descend from. We will also explore extensions, such as optimizations for read-only tables and vertical

partitioning mentioned in Section 3. It is a research task to see how far this approach can be pushed and how successful it will be.

In the meantime, horizontal partitioning and indexing options can be specified manually by a knowledgeable user.

4.4 Transaction Management, Replication and Recovery

Since H-Store implements two (or more) copies of each table, replicas must be transactionally updated. This is accomplished by directing each SQL read command to any replica and each SQL update to all replicas.

Moreover, every transaction receives a **timestamp** on entry to H-Store, which consists of a (site_id, local_unique_timestamp) pair. Given an ordering of sites, timestamps are unique and form a total order. We assume that the local clocks which generate local timestamps are kept nearly in sync with each other, using an algorithm like NTP [Mil89].

There are multiple situations which H-Store leverages to streamline concurrency control and commit protocols.

Single-sited/one shot: If all transaction classes are single-sited or one-shot, then individual transaction can be dispatched to the correct replica sites and executed to completion there. Unless all transaction classes are sterile, each execution site must wait a small period of time (meant to account for network delays) for transactions arriving from other initiators, so that the execution is in timestamp order. By increasing latency by a small amount, all replicas will by updated in the same order; in a local area network, maximum delays will be sub-millisecond. This will guarantee the identical outcome at each replica. Hence, data inconsistency between the replicas cannot occur. Also, all replicas will commit or all replicas will abort. Hence, each transaction can commit or abort locally, confident that the same outcome will occur at the other replicas. There is no redo log, no concurrency control, and no distributed commit processing.

Two-phase: No undo-log is required. Thus, if combined with the above properties, no transaction facilities are required at all.

Sterile: If all transaction classes are sterile, then execution can proceed normally with no concurrency control. Further, the need to issue timestamps and execute transactions in the same order on all replicas is obviated. However, if multiple sites are involved in query processing, then there is no guarantee that all sites will abort or all sites will continue. In this case, workers must respond "abort" or "continue" at the end of the first phase, and the execution supervisor must communicate this information to worker sites. Hence, standard commit distributed processing must be done at the end of phase one. This extra overhead can be avoided if the transaction is strongly two-phase.

Other cases: For other cases (non-sterile, non-single-sited, non one-shot), we need to endure the overhead of some sort of concurrency control scheme. All RDBMSs we are familiar with use dynamic locking to achieve transaction consistency. This decision followed pioneering simulation work in the 1980's [ACL87] that showed that locking worked better than other alternatives. However, we believe that dynamic locking is a poor choice for H-Store for the following reasons:

1) Transactions are very short-lived. There are no user-stalls and no disk activity. Hence, transactions are alive for very short time periods. This favors optimistic methods over pessimistic methods, like dynamic locking. Others, for example architects and programming language designers using transactions in memory models [HM93], have reached the same conclusion.
2) Every transaction is decomposed into collections of sub-commands, which are local to a given site. As noted earlier, the collection of sub commands are run in a single threaded fashion at each site. Again, this results in no latch waits, smaller total execution times, and again favors more optimistic methods.
3) We assume that we receive the entire collection of transaction classes in advance. This information can be used to advantage, as has been done previously by systems such as the SDD-1 scheme from the 1970's [BSR80] to reduce the concurrency control overhead.
4) In a well designed system there are **very few** transaction collisions and **very very few** deadlocks. These situations degrade performance and the workload is invariably modified by application designers to remove them. Hence, one should design for the "no collision" case, rather than using pessimistic methods.

The H-Store scheme takes advantage of these factors.

Every (non-sterile, non single-sited, non one-shot) transaction class has a collection of transaction classes with which it might conflict and arrives at some site in the grid and interacts with a transaction coordinator at that site. The transaction coordinator acts as the execution supervisor at the arrival site and sends out the subplan pieces to the various sites. A *worker* site receives a subplan and waits for the same small period of time mentioned above for other possibly conflicting transactions with lower timestamps to arrive. Then, the worker:

- Executes the subplan, if there is no uncommitted, potentially conflicting transaction at his site with a lower timestamp, and then sends his output data to the site requiring it, which may be an intermediate site or the transaction coordinator.
- Issues an abort to the coordinator otherwise

If the coordinator receives an "ok" from all sites, it continues with the transaction by issuing the next collection of subplans, perhaps with C++ logic interspersed. If there are no more subplans, then it commits the transaction. Otherwise, it aborts.

The above algorithm is the *basic* H-Store strategy. During execution, a transaction monitor watches the percentage of successful transactions. If there are too many aborts, H-Store dynamically moves to the following more sophisticated strategy.

Before executing or aborting the subplan, noted above, each worker site stalls by a length of time approximated by `MaxD * average_round_trip_message_delay` to see if a subplan with an earlier timestamp appears. If so, the worker site correctly sequences the subplans, thereby lowering the probability of abort. MaxD is the maximum depth of a conflicting transaction class.

This *intermediate* strategy lowers the abort probability, but at a cost of some number of msecs of increased latency. We are currently running simulations to demonstrate the circumstances under which this results in improved performance.

Our last *advanced* strategy keeps track of the read set and write set of each transaction at each site. In this case, a worker site runs each subplan, and then aborts the subplan if necessary according to standard optimistic concurrency control rules. At some extra overhead in bookkeeping and additional work discarded on aborts,

the probability of conflict can be further reduced. Again, simulations are in progress to determine when this is a winning strategy.

In summary, our H-Store concurrency control algorithm is:

- Run sterile, single-sited and one-shot transactions with no controls
- Other transactions are run with the basic strategy
- If there are too many aborts, escalate to the intermediate strategy
- If there are still too many aborts, further escalate to the advanced strategy.

It should be noted that this strategy is a sophisticated optimistic concurrency control scheme. Optimistic methods have been extensively investigated previously [KR81, ACL87]. Moreover, the Ants DBMS [Ants07] leverages commutativity to lower locking costs. Hence, this section should be considered as a very low overhead consolidation of known techniques.

Notice that we have not yet employed any sophisticated scheduling techniques to lower conflict. For example, it is possible to run examples from all pairs of transaction classes and record the conflict frequency. Then, a scheduler could take this information into account, and try to avoid running transactions together with a high probability of conflict.

The next section shows how these techniques and the rest of the H-Store design works on TPC-C.

5. A Performance Comparison

TPC-C runs on the schema diagramed in Figure 1, and contains 5 transaction classes (`new_order, payment, order status, delivery and stock_level`).

Because of space limitations, we will not include the code for these transactions; the interested reader is referred to the TPC-C specification [TPCC]. Table 1 summarizes their behavior.

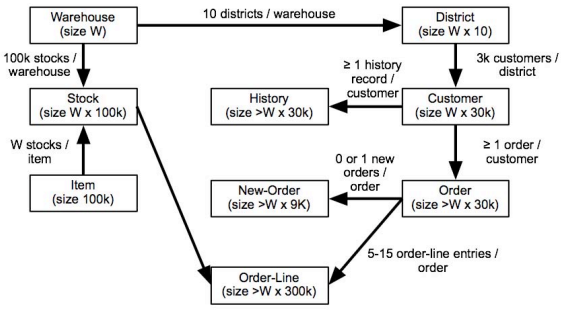

Figure 1: TPC-C Schema (reproduced from the TPC-C specification version 5.8.0, page 10)

There are three possible strategies for an efficient H-Store implementation of TPC-C. First, we could run on a single core, single CPU machine. This automatically makes every transaction class single-sited, and each transaction can be run to completion in a single-threaded environment. The paired-HA site will achieve the same execution order, since, as will be seen momentarily, all transaction classes can be made strongly two-phase, meaning that all transactions will either succeed at both sites or abort at both sites. Hence, on a single site with a paired HA site, ACID properties are achieved with no overhead whatsoever. The other two strategies are for parallel operation on multi-core and/or multi-CPUs systems. They involve making the workload either sterile or one-shot, which, as we discussed in the previous section, are sufficient to allow us to run queries without conventional concurrency control. To do this, we will need to perform some trickery with the TPC-C workload; before describing this, we first address data partitioning.

Table 1: TPC-C Transaction Classes

`new_order`	Place an order for a customer. 90% of all orders can be supplied in full by stocks from the customer's "home" warehouse; 10% need to access stock belonging to a remote warehouse. Read/write transaction. No minimum percentage of mix required, but about 50% of transactions are new_order transactions.
`payment`	Updates the customer's balance and warehouse/district sales fields. 85% of updates go to customer's home warehouse; 15% to a remote warehouse. Read/write transaction. Must be at least 43% of transaction mix.
`order_ status`	Queries the status of a customer's last order. Read only. Must be at least 4% of transaction mix.
`delivery`	Select a warehouse, and for each of 10 districts "deliver" an order, which means removing a record from the new-order table and updating the customer's account balance. Each delivery can be a separate transaction; Must be at least 4% of transaction mix.
`stock_ level`	Finds items with a stock level below a threshold; read only, must read committed data but does not need serializability. Must be at least 4% of transaction mix.

TPC-C is not a tree-structured schema. The presence of the Item table as well as the relationship of Order-line with Stock make it a non-tree schema. The Item table, however, is read-only and can be replicated at each site. The Order-line table can be partitioned according to Warehouse to each site. With such replication and partitioning, the schema is decomposed such that each site has a subset of the records rooted at a distinct partition of the warehouses. This will be termed the *basic* H-Store strategy for partitioning and replication.

5.1 Query classes

All transaction classes except `new_order` are already two-phase since they never need to abort. `New_order` may need to abort, since it is possible that its input contains invalid item numbers. However, it is permissible in the TPC-C specification to run a query for each item number at the beginning of the transaction to check for valid item numbers. By rearranging the transaction logic, all transaction classes become two-phase. It is also true that all transaction classes are strongly two-phase. This is because the Item table is never updated, and therefore all `new_order` transactions sent to all replicas always reach the same decision of whether to abort or not.

All 5 transaction classes appear to be sterile when considered with the basic partitioning and replication strategy. We make three observations in this regard.

First, the `new_order` transaction inserts a tuple in both the Orders table and New_Orders table as well as line items in the Line_order table. At each site, these operations will be part of a single sub-plan, and there will be no interleaved operations. This will ensure that the `order_status` transaction does not see

partially completed new orders. Second, because `new_order` and `payment` transactions in TPC-C are strongly two-phase, no additional coordination is needed between sites in the event that one of these transactions updates a "remote" warehouse relative to the customer making the order or payment.

Third, the `stock_level` transaction is allowed to run as multiple transactions which can see stock levels for different items at different points in time, as long as the stock level results from committed transactions. Because `new_orders` are aborted, if necessary, before they perform any updates, any stock information read comes from committed transactions (or transactions that will be committed soon).

Hence, all transaction classes can be made sterile and strongly two-phase. As such, they achieve a valid execution of TPC-C with no concurrency control. Although we could have tested this configuration, we decided to employ additional manipulation of the workload to also make all transaction classes one-shot, doing so improves performance.

With the basic strategy, all transaction classes, except `new_order` and `payment` are single-sited, and therefore one-shot. `Payment` is already one shot, since there is no need to exchange data when updating a remote warehouse. `New_order`, however, needs to insert in Order-line information about the district of a stock entry which may reside in a remote site. Since that field is never updated, and there are no deletes/inserts into the Stock table, we can vertically partition Stock and replicate the read-only parts of it across all sites. With this replication trick added to the basic strategy, `new_order` becomes one shot.

As a result, with the basic strategy augmented with the tricks described above, all transaction classes become one-shot and strongly two-phase. As long as we add a short delay as mentioned in Section 4.4, ACID properties are achieved with no concurrency control overhead whatsoever. This is the configuration on which benchmark results are reported in Section 5.3

It is difficult to imagine that an automatic program could figure out what is required to make TPC-C either one-shot or sterile. Hence, a knowledgeable human would have to carefully code the transactions classes. It is likely, however, that most transaction classes will be simpler to analyze. As such, it is an open question how successful automatic transaction class analysis will be.

5.2 Implementation

We implemented a variant of TPC-C on H-Store and on a very popular commercial RDBMS. The same driver was used for both systems and generated transactions at the maximum rate without modeling think time. These transactions were delivered to both systems using TCP/IP. All transaction classes were implemented as stored procedures. In H-Store the transaction logic was coded in C++, with local procedure calls to H-Store query execution. In contrast, the transaction logic for the commercial system was written using their proprietary stored procedure language. High availability and communication with user terminals was not included for either system.

Both DBMSs were run on a dual-core 2.8GHz CPU computer system, with 4 Gbytes of main memory and four 250 GB SATA disk drives. Both DBMSs used horizontal partitioning to advantage.

5.3 Results

On this configuration, H-Store ran 70,416 TPC-C transactions per second. In contrast, we could only coax 850 transactions per second from the commercial system, in spite of several days of tuning by a professional DBA, who specializes in this vendor's product. Hence, H-Store ran a factor of 82 faster (almost two orders of magnitude).

Per our earlier discussion, the bottleneck for the commercial system was logging overhead. That system spent about 2/3 of its total elapsed time inside the logging system. One of us spent many hours trying to tune the logging system (log to a dedicated disk, change the size of the group commit; all to no avail). If logging was turned off completely, and assuming no other bottleneck creeps up, then throughput would increase to about 2,500 transactions per second.

The next bottleneck appears to be the concurrency control system. In future experiments, we plan to tease apart the overhead contributions which result from:

- Redo logging
- Undo logging
- Latching
- Locking

Finally, though we did not implement all of the TPC-C specification (we did not, for example, model wait times), it is also instructive to compare our partial TPC-C implementation with TPC-C performance records on the TPC website[2]. The highest performing TPC-C implementation executes about 4 million new-order transactions per minute, or a total of about 133,000 total transactions per second. This is on a 128 core shared memory machine, so this implementation is getting about 1000 transactions per core. Contrast this with 400 transactions per core in our benchmark on a commercial system on a (rather pokey) desktop machine, or 35,000 transactions per core in H-Store! Also, note that H-Store is within a factor of two of the best TPC-C results on a machine costing around $1000.00

In summary, the conclusion to be reached is that nearly two orders of magnitude in performance improvement are available to a system designed along the lines of H-Store.

6. Some Comments about a "One Size Does Not Fit All" World

If the results of this paper are to be believed, then we are heading toward a world with at least 5 (and probably more) specialized engines and the death of the "one size fits all" legacy systems. This section considers some of the consequences of such an architectural shift.

6.1 The Relational Model Is not Necessarily the Answer

Having survived the great debate of 1974 [Rus74] and the surrounding arguments between the advocates of the Codasyl and relational models, we are reluctant to bring up this particular "sacred cow". However, it seems appropriate to consider the data model (or data models) that we build systems around. In the 1970's the DBMS world contained only business data processing applications, and Ted Codd's idea of normalizing data into flat

[2] http://www.tcp.org/tpcc/results/tpcc_perf_results.asp

tables has served our community well over the subsequent 30 years. However, there are now other markets, whose needs must be considered. These include data warehouses, web-oriented search, real-time analytics, and semi-structured data markets.

We offer the following observations.

1. In the data warehouse market, nearly 100% of all schemas are stars or snowflakes, containing a central fact table with 1-n joins to surrounding dimension tables, which may in turn participate in further 1-n joins to second level dimension tables, and so forth. Although stars and snowflakes are easily modeled using relational schemas, in fact, an entity-relationship model would be simpler in this environment and more natural. Moreover, warehouse queries would be simpler in an E-R model. Lastly, warehouse operations that are incredibly expensive with a relational implementation, for example changing the key of a row in a dimension table, might be made faster with some sort of E-R implementation.
2. In the stream processing market, there is a need to:
    a. Process streams of messages at high speed
    b. Correlate such streams with stored data

    To accomplish both tasks, there is widespread enthusiasm for StreamSQL, a generalization of SQL that allows a programmer to mix stored tables and streams in the FROM clause of a SQL statement. This work has evolved from the pioneering work of the Stanford Stream group [ABW06] and is being actively discussed for standardization. Of course, StreamSQL supports relational schemas for both tables and streams.

    However, commercial feeds, such as Reuters, Infodyne, etc., have all chosen some data model for their messages to obey. Some are flat and fit nicely into a relational schema. However, several are hierarchical, such as the FX feed for foreign exchange. Stream processing systems, such as StreamBase and Coral8, currently support only flat (relational) messages. In such systems, a front-end adaptor must normalize hierarchical objects into several flat message types for processing. Unfortunately, it is rather painful to join the constituent pieces of a source message back together when processing on multiple parts of a hierarchy is necessary.

    To solve this problem, we expect the stream processing vendors to move aggressively to hierarchical data models. Hence, they will assuredly deviate from Ted Codd's principles.
3. Text processing obviously has never used a relational model.
4. Any scientific-oriented DBMS, such as ASAP [SBC+07], will probably implement arrays, not tables as their basic data type.
5. There has recently been considerable debate over good data models for semi-structured data. There is certainly fierce debate over the excessive complexity of XMLSchema [SC05]. There are fans of using RDF for such data [MM04], and some who argue that RDF can be efficiently implemented by a relational column store [AMM+07]. Suffice it to say that there are many ideas on which way to go in this area.

In summary, the relational model was developed for a "one size fits all" world. The various specialized systems which we envision can each rethink what data model would work best for their particular needs.

6.2 SQL is Not the Answer

SQL is a "one size fits all" language. In an OLTP world one never asks for the employees who earn more than their managers. In fact, there are no ad-hoc queries, as noted earlier. Hence, one can implement a smaller language than SQL. For performance reasons, stored procedures are omni-present. In a data warehouse world, one needs a different subset of SQL, since there are complex ad-hoc queries, but no stored procedures. Hence, the various storage engines can implement vertical-market specific languages, which will be simpler than the daunting complexity of SQL.

Rethinking how many query languages should exist as well as their complexity will have a huge side benefit. At this point SQL is a legacy language with many known serious flaws, as noted by Chris Date two decades ago [Dat84]. Next time around, we can do a better job.

When rethinking data access languages, we are reminded of a raging discussion from the 1970's. On the one-hand, there were advocates of a data sublanguage, which could be interfaced to any programming language. This has led to high overhead interfaces, such as JDBC and ODBC. In addition, these interfaces are very difficult to use from a conventional programming language.

In contrast, some members of the DBMS community proposed much nicer embedding of database capabilities in programming languages, typified in the 1970s by Pascal R [Sch80] and Rigel [RS79]. Both had clean integration with programming language facilities, such as control flow, local variables, etc. Chris Date also proposed an extension to PL/1 with the same purpose [Dat76].

Obviously none of these languages ever caught on, and the data sublanguage camp prevailed. The couplings between a programming language and a data sublanguage that our community has designed are ugly beyond belief and are low productivity systems that date from a different era. Hence, we advocate scrapping sublanguages completely, in favor of much cleaner language embeddings.

In the programming language community, there has been an explosion of "little languages" such as Python, Perl, Ruby and PHP. The idea is that one should use the best language available for any particular task at hand. Also little languages are attractive because they are easier to learn than general purpose languages. From afar, this phenomenon appears to be the death of "one size fits all" in the programming language world.

Little languages have two very desirable properties. First, they are mostly open source, and can be altered by the community. Second they are less daunting to modify than the current general purpose languages. As such, we are advocates of modifying little languages to include clean embeddings of DBMS access.

Our current favorite example of this approach is Ruby-on-Rails[3]. This system is the little language, Ruby, extended with integrated support for database access and manipulation through the "model-view-controller" programming pattern.. Ruby-on-Rails compiles into standard JDBC, but hides all the complexity of that interface.

Hence, H-Store plans to move from C++ to Ruby-on-Rails as our stored procedure language. Of course, the language run-time must be linked into the DBMS address space, and must be altered

[3] http://www.rubyonrails.org

to make calls to DBMS services using high performance local procedure calls, not JDBC.

7. Summary and Future Work

In the last quarter of a century, there has been a dramatic shift in:

1. DBMS markets: from business data processing to a collection of markets, with varying requirements
2. Necessary features: new requirements include shared nothing support and high availability
3. Technology: large main memories, the possibility of hot standbys, and the web change most everything

The result is:

1. The predicted demise of "one size fits all"
2. The inappropriateness of current relational implementations for any segment of the market
3. The necessity of rethinking both data models and query languages for the specialized engines, which we expect to be dominant in the various vertical markets

Our H-Store prototype demonstrates the performance gains that can be had when this conventional thinking is questioned. Of course, beyond these encouraging initial performance results, there are a number of areas where future work is needed. In particular:

- More work is needed to identify when it is possible to automatically identify single-sited, two-phase, and one-shot applications. "Auto-everything" tools that can suggest partitions that lead to these properties are also essential.
- The rise of multi-core machines suggests that there may be interesting optimizations related to sharing of work between logical sites physically co-located on the same machine.
- A careful study of the performance of the various transaction management strategies outlined in Section 3 is needed.
- A study of the overheads of the various components of a OLTP system – logging, transaction processing and two-phase commit, locking, JDBC/ODBC, etc -- would help identify which aspects of traditional DBMS design contribute most to the overheads we have observed.
- After stripping out all of these overheads, our H-Store implementation is now limited by the performance of in-memory data structures, suggesting that optimizing these structures will be important. For example, we found that the simple optimization of representing read-only tables as arrays offered significant gains in transaction throughput in our H-Store implementation.
- Integration with data warehousing tools – for example, by using no-overwrite storage and occasionally dumping records into a warehouse – will be essential if H-Store-like systems are to seamlessly co-exist with data warehouses.

In short, the current situation in the DBMS community reminds us of the period 1970 - 1985 where there was a "group grope" for the best way to build DBMS engines and dramatic changes in commercial products and DBMS vendors ensued. The 1970 - 1985 period was a time of intense debate, a myriad of ideas, and considerable upheaval.

We predict the next fifteen years will have the same feel.

References

[ABW06] A. Arasu, S. Babu, and J. Widom. "The CQL Continuous Query Language: Semantic Foundations and Query Execution." *The VLDB Journal, 15(2)*, June 2006.

[ACL87] Agrawal, R., Carey, M. J., and Livny, M. "Concurrency control performance modeling: alternatives and implications." *ACM Trans. Database Syst. 12(4)*, Nov. 1987.

[AMM+07] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. "Scalable Semantic Web Data Management Using Vertical Partitioning." In *Proc. VLDB*, 2007.

[Ants07] ANTs Software. ANTs Data Server - Technical White Paper, http://www.ants.com, 2007.

[BSR80] Bernstein, P.A., Shipman, D., and Rothnie, J. B. "Concurrency Control in a System for Distributed Databases (SDD-1)." *ACM Trans. Database Syst. 5(1)*, March 1980.

[Bon02] P. A. Boncz. "Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications." Ph.D. Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[Dat76] C. J. Date. "An Architecture for High-Level Language Database Extensions." In *Proc. SIGMOD*, 1976.

[Dat84] Date, C. J. "A critique of the SQL database language." In *SIGMOD Record 14(3):8-54*, Nov. 1984.

[DGS+90] Dewitt, D. J., Ghandeharizadeh, S., Schneider, D. A., Bricker, A., Hsiao, H., and Rasmussen, R. "The Gamma Database Machine Project." *IEEE Transactions on Knowledge and Data Engineering 2(1):44-62*, March 1990.

[Hel07] P. Helland. "Life beyond Distributed Transactions: an Apostate's Opinion." In *Proc. CIDR*, 2007.

[HM93] Herlihy, M. and Moss, J. E. "Transactional memory: architectural support for lock-free data structures." In *Proc. ISCA*, 1993.

[KL81] Kung, H. T. and Robinson, J. T. "On optimistic methods for concurrency control." *ACM Trans. Database Syst. 6(2):213-226*, June 1981.

[LM06] E. Lau and S. Madden. "An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse." In *Proc. VLDB*, 2006.

[MHL+92] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging." *ACM Trans. Database Syst. 17(1):94-162*, March 1992.

[Mil89] Mills, D. L. "On the Accuracy and Stability of Clocks Synchronized by the Network Time Protocol in the Internet System." *SIGCOMM Comput. Commun. Rev. 20(1):65-75*, Dec. 1989.

[MM04] Manola, F. and Miller, E. (eds). RDF Primer. W3C Specification, February 10, 2004. http://www.w3.org/TR/REC-rdf-primer-20040210/

[RR99] Rao, J. and Ross, K. A. "Cache Conscious Indexing for Decision-Support in Main Memory." In *Proc. VLDB*, 1999.

[RR00] Rao, J. and Ross, K. A. "Making B+- trees cache conscious in main memory." In *SIGMOD Record, 29(2):475-486*, June 2000.

[RS79] L. A. Rowe and K. A. Shoens. "Data Abstractions, Views and Updates in RIGEL." In *Proc. SIGMOD*, 1979.

[Rus74] Randall Rustin (Ed.): Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes.

[SAB+05] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. "C-Store: A Column-oriented DBMS." In *Proc. VLDB*, 2005.

[SBC+07] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. "One Size Fits All? - Part 2: Benchmarking Results." In *Proc. CIDR*, 2007.

[SC05] M. Stonebraker and U. Cetintemel. "One Size Fits All: An Idea whose Time has Come and Gone." In *Proc. ICDE*, 2005.

[Sch80] Schmidt, J.W. et al. "Pascal/R Report." U Hamburg, Fachbereich Informatik, Report 66, Jan 1980.

[TPCC] The Transaction Processing Council. TPC-C Benchmark (Revision 5.8.0), 2006.
http://www.tpc.org/tpcc/spec/tpcc_current.pdf