"One Size Fits All": An Idea Whose Time Has Come and Gone

Michael Stonebraker
Computer Science and Artificial Intelligence Laboratory, M.I.T., and StreamBase Systems, Inc.
stonebraker@csail.mit.edu

Uğur Çetintemel
Department of Computer Science Brown University, and StreamBase Systems, Inc.
ugur@cs.brown.edu

Abstract

The last 25 years of commercial DBMS development can be summed up in a single phrase: "One size fits all". This phrase refers to the fact that the traditional DBMS architecture (originally designed and optimized for business data processing) has been used to support many data-centric applications with widely varying characteristics and requirements.

In this paper, we argue that this concept is no longer applicable to the database market, and that the commercial world will fracture into a collection of independent database engines, some of which may be unified by a common front-end parser. We use examples from the stream-processing market and the data-warehouse market to bolster our claims. We also briefly discuss other markets for which the traditional architecture is a poor fit and argue for a critical rethinking of the current factoring of systems services into products.

1. Introduction

Relational DBMSs arrived on the scene as research prototypes in the 1970's, in the form of System R [10] and INGRES [27]. The main thrust of both prototypes was to surpass IMS in value to customers on the applications that IMS was used for, namely "business data processing". Hence, both systems were architected for on-line transaction processing (OLTP) applications, and their commercial counterparts (i.e., DB2 and INGRES, respectively) found acceptance in this arena in the 1980's. Other vendors (e.g., Sybase, Oracle, and Informix) followed the same basic DBMS model, which stores relational tables row-by-row, uses B-trees for indexing, uses a cost-based optimizer, and provides ACID transaction properties.

Since the early 1980's, the major DBMS vendors have steadfastly stuck to a "one size fits all" strategy, whereby they maintain a single code line with all DBMS services. The reasons for this choice are straightforward — the use of multiple code lines causes various practical problems, including:

- *a cost problem*, because maintenance costs increase at least linearly with the number of code lines;
- *a compatibility problem*, because all applications have to run against every code line;
- *a sales problem*, because salespeople get confused about which product to try to sell to a customer; and
- *a marketing problem*, because multiple code lines need to be positioned correctly in the marketplace.

To avoid these problems, all the major DBMS vendors have followed the adage "put all wood behind one arrowhead". In this paper we argue that this strategy has failed already, and will fail more dramatically off into the future.

The rest of the paper is structured as follows. In Section 2, we briefly indicate why the single code-line strategy has failed already by citing some of the key characteristics of the data warehouse market. In Section 3, we discuss stream processing applications and indicate a particular example where a specialized stream processing engine outperforms an RDBMS by two orders of magnitude. Section 4 then turns to the reasons for the performance difference, and indicates that DBMS technology is not likely to be able to adapt to be competitive in this market. Hence, we expect stream processing engines to thrive in the marketplace. In Section 5, we discuss a collection of other markets where one size is not likely to fit all, and other specialized database systems may be feasible. Hence, the fragmentation of the DBMS market may be fairly extensive. In Section 6, we offer some comments about the factoring of system software into products. Finally, we close the paper with some concluding remarks in Section 7.

2. Data warehousing

In the early 1990's, a new trend appeared: Enterprises wanted to gather together data from multiple operational databases into a data warehouse for business intelligence

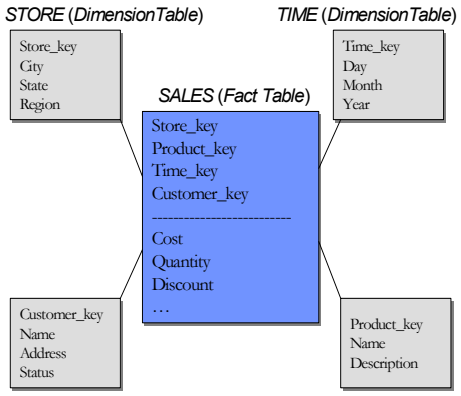

Figure 1: A typical star schema

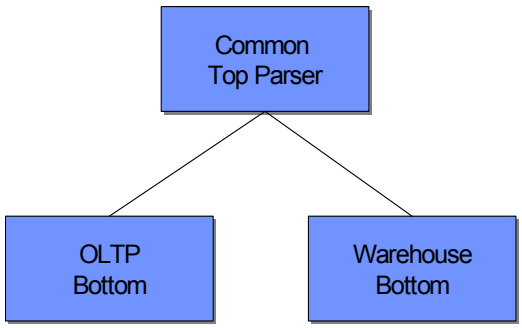

Figure 2: The architecture of current DBMSs

purposes. A typical large enterprise has 50 or so operational systems, each with an on-line user community who expect fast response time. System administrators were (and still are) reluctant to allow business-intelligence users onto the same systems, fearing that the complex ad-hoc queries from these users will degrade response time for the on-line community. In addition, business-intelligence users often want to see historical trends, as well as correlate data from multiple operational databases. These features are very different from those required by on-line users.

For these reasons, essentially every enterprise created a large data warehouse, and periodically "scraped" the data from operational systems into it. Business-intelligence users could then run their complex ad-hoc queries against the data in the warehouse, without affecting the on-line users. Although most warehouse projects were dramatically over budget and ended up delivering only a subset of promised functionality, they still delivered a reasonable return on investment. In fact, it is widely acknowledged that historical warehouses of retail transactions pay for themselves within a year, primarily as a result of more informed stock rotation and buying decisions. For example, a business-intelligence user can discover that pet rocks are out and Barbie dolls are in, and then make appropriate merchandise placement and buying decisions.

Data warehouses are very different from OLTP systems. OLTP systems have been optimized for updates, as the main business activity is typically to sell a good or service. In contrast, the main activity in data warehouses is ad-hoc queries, which are often quite complex. Hence, periodic load of new data interspersed with ad-hoc query activity is what a typical warehouse experiences.

The standard wisdom in data warehouse schemas is to create a fact table, containing the "who, what, when, where" about each operational transaction. For example, Figure 1 shows the schema for a typical retailer. Note the central fact table, which holds an entry for each item that is scanned by a cashier in each store in its chain. In addition, the warehouse contains dimension tables, with information on each store, each customer, each product, and each time period. In effect, the fact table contains a foreign key for each of these dimensions, and a star schema is the natural result. Such star schemas are omni-present in warehouse environments, but are virtually non-existent in OLTP environments.

It is a well known homily that warehouse applications run much better using bit-map indexes while OLTP users prefer B-tree indexes. The reasons are straightforward: bit-map indexes are faster and more compact on warehouse workloads, while failing to work well in OLTP environments. As a result, many vendors support both B-tree indexes and bit-map indexes in their DBMS products.

In addition, materialized views are a useful optimization tactic in warehouse worlds, but never in OLTP worlds. In contrast, normal ("virtual") views find acceptance in OLTP environments.

To a first approximation, most vendors have a warehouse DBMS (bit-map indexes, materialized views, star schemas and optimizer tactics for star schema queries) and an OLTP DBMS (B-tree indexes and a standard cost-based optimizer), which are united by a common parser, as illustrated in Figure 2.

Although this configuration allows such a vendor to market his DBMS product as a single system, because of the single user interface, in effect, she is selling multiple systems. Moreover, there will considerable pressure from both the OLTP and warehouse markets for features that are of no use in the other world. For example, it is common practice in OLTP databases to represent the state (in the United States) portion of an address as a two-byte character string. In contrast, it is obvious that 50 states can be coded into six bits. If there are enough queries and enough data to justify the cost of coding the state field, then the later representation is advantageous. This is usually true in warehouses and never true in OLTP. Hence, elaborate coding of fields will be a warehouse feature that has little or no utility in OLTP. The inclusion of additional market-specific features will make commercial products look increasingly like the architecture illustrated in Figure 2.

The illusion of "one size fits all" can be preserved as a marketing fiction for the two different systems of Figure

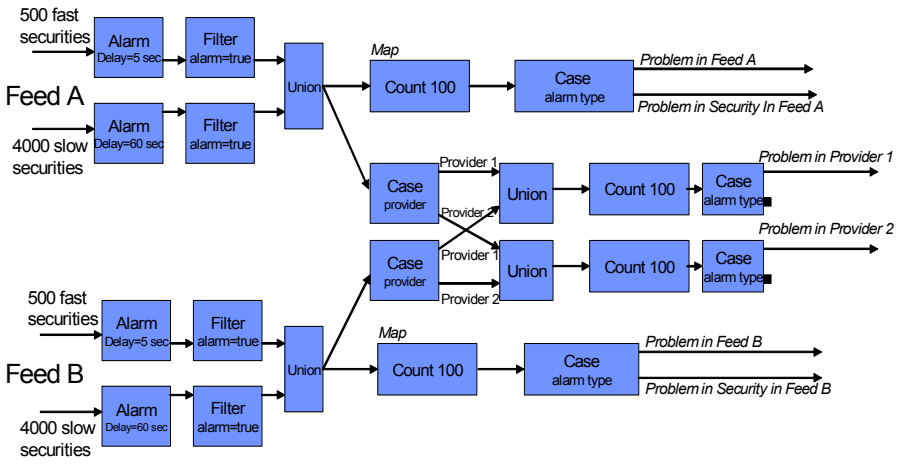

Figure 3: The Feed Alarm application in StreamBase

2, because of the common user interface. In the stream processing market, to which we now turn, such a common front end is impractical. Hence, not only will there be different engines but also different front ends. The marketing fiction of "one size fits all" will not fly in this world.

3. Stream processing

Recently, there has been considerable interest in the research community in stream processing applications [7, 13, 14, 20]. This interest is motivated by the upcoming commercial viability of sensor networks over the next few years. Although RFID has gotten all the press recently and will find widespread acceptance in retail applications dealing with supply chain optimization, there are many other technologies as well (e.g., Lojack [3]). Many industry pundits see a "green field" of monitoring applications that will be enabled by this "sea change" caused by networks of low-cost sensor devices.

3.1 Emerging sensor-based applications

There are obvious applications of sensor network technology in the military domain. For example, the US Army is investigating putting vital-signs monitors on all soldiers, so that they can optimize medical triage in combat situations. In addition, there is already a GPS system in many military vehicles, but it is not connected yet into a closed-loop system. Instead, the army would like to monitor the position of all vehicles and determine, in real time, if they are off course. Additionally, they would like a sensor on the gun turret; together with location, this will allow the detection of crossfire situations. A sensor on the gas gauge will allow the optimization of refueling. In all, an army battalion of 30,000 humans and 12,000 vehicles will soon be a large-scale sensor network of several hundred thousand nodes delivering state and position information in real time.

Processing nodes in the network and downstream servers must be capable of dealing with this "firehose" of data. Required operations include sophisticated alerting, such as the platoon commander wishes to know when three of his four vehicles cross the front line. Also required are historical queries, such as "Where has vehicle 12 been for the last two hours?" Lastly, requirements encompass longitudinal queries, such as "What is the overall state of readiness of the force right now?"

Other sensor-based monitoring applications will also come over time in many non-military applications. Monitoring traffic congestion and suggesting alternate travel routes is one example. A related application is variable, congestion-based tolling on highway systems, which was the inspiration behind the Linear Road benchmark [9]. Amusement parks will soon turn passive wristbands on customers into active sensors, so that rides can be optimized and lost children located. Cell phones are already active devices, and one can easily imagine a service whereby the closest restaurant to a hungry customer can be located. Even library books will be sensor tagged, because if one is mis-shelved, it may be lost forever in a big library.

There is widespread speculation that conventional DBMSs will not perform well on this new class of monitoring applications. In fact, on Linear Road, traditional solutions are nearly an order of magnitude slower than a special purpose stream processing engine [9]. The inapplicability of the traditional DBMS technology to streaming applications is also bolstered by an examination of the current application areas with streaming data. We now discuss our experience with such an application, financial-feed processing.

3.2 An existing application: financial-feed processing

Most large financial institutions subscribe to feeds that deliver real-time data on market activity, specifically news, consummated trades, bids and asks, etc. Reuters, Bloomberg and Infodyne are examples of vendors that

deliver such feeds. Financial institutions have a variety of applications that process such feeds. These include systems that produce real-time business analytics, ones that perform electronic trading, ones that ensure legal compliance of all trades to the various company and SEC rules, and ones that compute real-time risk and market exposure to fluctuations in foreign exchange rates. The technology used to implement this class of applications is invariably "roll your own", because application experts have not had good luck with off-the-shelf system software products.

In order to explore feed processing issues more deeply, we now describe in detail a specific prototype application, which was specified by a large mutual fund company. This company subscribes to several commercial feeds, and has a current production application that watches all feeds for the presence of late data. The idea is to alert the traders if one of the commercial feeds is delayed, so that the traders can know not to trust the information provided by that feed. This company is unhappy with the performance and flexibility of their "roll your own" solution and requested a pilot using a stream processing engine.

The company engineers specified a simplified version of their current application to explore the performance differences between their current system and a stream processing engine. According to their specification, they were looking for maximum message processing throughput on a single PC-class machine for a subset of their application, which consisted of two feeds reporting data from two exchanges.

Specifically, there are 4500 securities, 500 of which are "fast moving". A stock tick on one of these securities is late if it occurs more than five seconds after the previous tick from the same security. The other 4000 symbols are slow moving, and a tick is late if 60 seconds have elapsed since the previous tick.

There are two feed providers and the company wished to receive an alert message each time there is a late tick from either provider. In addition, they wished to maintain a counter for each provider. When 100 late ticks have been received from either provider, they wished to receive a special "this is really bad" message and then to suppress the subsequent individual tick reports

The last wrinkle in the company's specification was that they wished to accumulate late ticks from each of two exchanges, say NYSE and NASD, regardless of which feed vendor produced the late data. If 100 late messages were received from either exchange through either feed vendor, they wished to receive two additional special messages. In summary, they want four counters, each counting to 100, with a resulting special message. An abstract representation of the query diagram for this task is shown in Figure 3.

Although this prototype application is only a subset of the application logic used in the real production system, it represents a simple-to-specify task on which performance can be readily measured; as such, it is a representative example. We now turn to the speed of this example application on a stream processing engine as well as an RDBMS.

4. Performance discussion

The example application discussed in the previous section was implemented in the StreamBase stream processing engine (SPE) [5], which is basically a commercial, industrial-strength version of Aurora [8, 13]. On a 2.8Ghz Pentium processor with 512 Mbytes of memory and a single SCSI disk, the workflow in Figure 3 can be executed at 160,000 messages per second, before CPU saturation is observed. In contrast, StreamBase engineers could only coax 900 messages per second from an implementation of the same application using a popular commercial relational DBMS.

In this section, we discuss the main reasons that result in the two orders of magnitude difference in observed performance. As we argue below, the reasons have to do with the inbound processing model, correct primitives for stream processing, and seamless integration of DBMS processing with application processing. In addition, we also consider transactional behavior, which is often another major consideration.

4.1 "Inbound" versus "outbound" processing

Built fundamentally into the DBMS model of the world is what we term "outbound" processing, illustrated in Figure 4. Specifically, one inserts data into a database as a first step (step 1). After indexing the data and committing the transaction, that data is available for subsequent query processing (step 2) after which results are presented to the user (step 3). This model of "process-after-store" is at the heart of all conventional DBMSs, which is hardly surprising because, after all, the main function of a DBMS is to accept and then never lose data.

In real-time applications, the storage operation, which must occur before processing, adds significantly both to the delay (i.e., latency) in the application, as well as to the processing cost per message of the application. An alternative processing model that avoids this storage bottleneck is shown graphically in Figure 5. Here, input streams are pushed to the system (step 1) and get processed (step 2) as they "fly by" in memory by the query network. The results are then pushed to the client application(s) for consumption (step 3). Reads or writes to storage are optional and can be executed asynchronously in many cases, when they are present. The fact that storage is absent or optional saves both on cost and latency, resulting in significantly higher performance. This model, called "inbound" processing, is what is employed by a stream processing engine such as StreamBase.

One is, of course, led to ask "Can a DBMS do inbound processing?" DBMSs were originally designed as outbound processing engines, but grafted triggers onto their engines as an afterthought many years later. There are many restrictions on triggers (e.g., the number

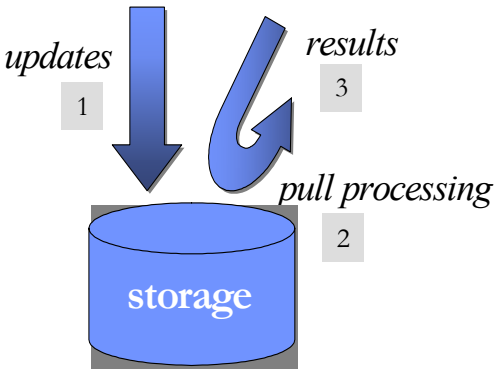

Figure 4: "Outbound" processing

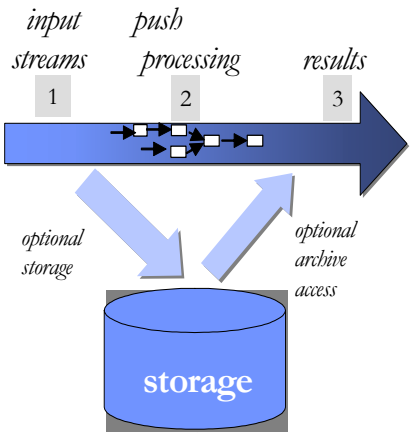

Figure 5: "Inbound" processing

allowed per table) and no way to ensure trigger safety (i.e., ensuring that triggers do not go into an infinite loop). Overall, there is very little or no programming support for triggers. For example, there is no way to see what triggers are in place in an application, and no way to add a trigger to a table through a graphical user interface. Moreover, virtual views and materialized views are provided for regular tables, but not for triggers. Lastly, triggers often have performance problems in existing engines. When StreamBase engineers tried to use them for the feed alarm application, they still could not obtain more than 900 messages per second. In summary, triggers are incorporated to the existing designs as an afterthought and are thus second-class citizens in current systems.

As such, relational DBMSs are outbound engines onto which limited inbound processing has been grafted. In contrast, stream processing engines, such as Aurora and StreamBase are fundamentally inbound processing engines. From the ground up, an inbound engine looks radically different from an outbound engine. For example, an outbound engine uses a "pull" model of processing, i.e., a query is submitted and it is the job of the engine to efficiently pull records out of storage to satisfy the query. In contrast, an inbound engine uses a "push" model of processing, and it is the job of the engine to efficiently push incoming messages through the processing steps entailed in the application.

Another way to view the distinction is that an outbound engine stores the data and then executes the queries against the data. In contrast, an inbound engine stores the queries and then passes the incoming data (messages) through the queries.

Although it seems conceivable to construct an engine that is either an inbound or an outbound engine, such a design is clearly a research project. In the meantime, DBMSs are optimized for outbound processing, and stream processing engines for inbound processing. In the feed alarm application, this difference in philosophy accounts for a substantial portion of the performance difference observed.

4.2 The correct primitives

SQL systems contain a sophisticated aggregation system, whereby a user can run a statistical computation over groupings of the records from a table in a database. The standard example is:

```
Select avg (salary)
From employee
Group by department
```

When the execution engine processes the last record in the table, it can emit the aggregate calculation for each group of records. However, this construct is of little benefit in streaming applications, where streams continue forever and there is no notion of "end of table".

Consequently, stream processing engines extend SQL (or some other aggregation language) with the notion of time windows. In StreamBase, windows can be defined based on clock time, number of messages, or breakpoints in some other attribute. In the feed alarm application, the leftmost box in each stream is such an aggregate box. The aggregate groups stocks by symbol and then defines windows to be ticks 1 and 2, 2 and 3, 3 and 4, etc. for each stock. Such "sliding windows" are often very useful in real-time applications.

In addition, StreamBase aggregates have been constructed to deal intelligently with messages which are late, out-of-order, or missing. In the feed alarm application, the customer is fundamentally interested in looking for late data. StreamBase allows aggregates on windows to have two additional parameters. The first is a *timeout* parameter, which instructs the StreamBase execution engine to close a window and emit a value even if the condition for closing the window has not been satisfied. This parameter effectively deals with late or missing tuples. The second parameter is *slack*, which is a directive to the execution engine to keep a window open, after its closing condition has been satisfied. This parameter addresses disorder in tuple arrivals. These two parameters allow the user to specify how to deal with

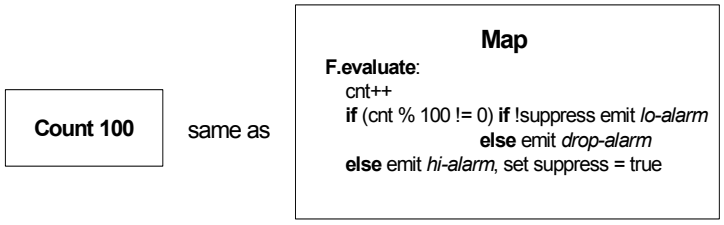

Figure 6: "Count100" logic

stream abnormalities and can be effectively utilized to improve system resilience.

In the feed alarm application each window is two ticks, but has a timeout of either 5 or 60 seconds. This will cause windows to be closed if the inter-arrival time between successive ticks exceeds the maximum defined by the user. This is a very efficient way to discover late data; i.e., as a side effect of the highly-tuned aggregate logic. In the example application, the box after each aggregate discards the valid data and keeps only the timeout messages. The remainder of the application performs the necessary bookkeeping on these timeouts.

Having the right primitives at the lower layers of the system enables very high performance. In contrast, a relational engine contains no such built-in constructs. Simulating their effect with conventional SQL is quite tedious, and results in a second significant difference in performance.

It is possible to add time windows to SQL, but these make no sense on stored data. Hence, window constructs would have to be integrated into some sort of an inbound processing model.

4.3 Seamless integration of DBMS processing and application logic

Relational DBMSs were all designed to have client-server architectures. In this model, there are many client applications, which can be written by arbitrary people, and which are therefore typically untrusted. Hence, for security and reliability reasons, these client applications are run in a separate address space from the DBMS. The cost of this choice is that the application runs in one address space while DBMS processing occurs in another, and a process switch is required to move from one address space to the other.

In contrast, the feed alarm application is an example of an embedded system. It is written by one person or group, who is trusted to "do the right thing". The entire application consists of (1) DBMS processing—for example the aggregation and filter boxes, (2) control logic to direct messages to the correct next processing step, and (3) application logic. In StreamBase, these three kinds of functionality can be freely interspersed. Application logic is supported with user-defined boxes, the Count100 box in our example financial-feed processing application. The actual code, shown in Figure 6, consists of four lines of C++ that counts to 100 and sets a flag that ensures that the correct messages are emitted. Control logic is supported by allowing multiple predicates in a filter box, and thereby multiple exit arcs. As such, a filter box performs "if-then-else" logic in addition to filtering streams.

In effect, the feed alarm application is a mix of DBMS-style processing, conditional expressions, and user-defined functions in a conventional programming language. This combination is performed by StreamBase within a single address space without any process switches. Such a seamless integration of DBMS logic with conventional programming facilities was proposed many years ago in Rigel [23] and Pascal-R [25], but was never implemented in commercial relational systems. Instead, the major vendors implemented stored procedures, which are much more limited programming systems. More recently, the emergence of object-relational engines provided blades or extenders, which are more powerful than stored procedures, but still do not facilitate flexible control logic.

Embedded systems do not need the protection provided by client-server DBMSs, and a two-tier architecture merely generates overhead. This is a third source of the performance difference observed in our example application.

Another integration issue, not exemplified by the feed alarm example, is the storage of state information in streaming applications. Most stream processing applications require saving some state, anywhere from modest numbers of megabytes to small numbers of gigabytes. Such state information may include (1) reference data (i.e., what stocks are of interest), (2) translation tables (in case feeds use different symbols for the same stock), and (3) historical data (e.g., "how many late ticks were observed every day during the last year?"). As such, tabular storage of data is a requirement for most stream processing applications.

StreamBase embeds BerkeleyDB [4] for state storage. However, there is approximately one order of magnitude performance difference between calling BerkeleyDB in the StreamBase address space and calling it in client-server mode in a different address space. This is yet another reason to avoid process switches by mixing DBMS and application processing in one address space.

Although one might suggest that DBMSs enhance their programming models to address this performance problem, there are very good reasons why client-server DBMSs were designed the way they are. Most business data processing applications need the protection that is afforded by this model. Stored procedures and object-relational blades were an attempt to move some of the client logic into the server to gain performance. To move further, a DBMS would have to implement both an embedded and a non-embedded model, with different run time systems. Again, this would amount to giving up on "one size fits all".

In contrast, feed processing systems are invariably embedded applications. Hence, the application and the DBMS are written by the same people, and driven from external feeds, not from human-entered transactions. As such, there is no reason to protect the DBMS from the application, and it is perfectly acceptable to run both in the same address space. In an embedded processing model, it is reasonable to freely mix application logic, control logic and DBMS logic, which is exactly what StreamBase does.

4.4 High availability

It is a requirement of many stream-based applications to have high availability (HA) and stay up 7x24. Standard DBMS logging and crash recovery mechanisms (e.g., [22]) are ill-suited for the streaming world as they introduce several key problems.

First, log-based recovery may take large number of seconds to small numbers of minutes. During this period, the application would be "down". Such behavior is clearly undesirable in many real-time streaming domains (e.g., financial services). Second, in case of a crash, some effort must be made to buffer the incoming data streams, as otherwise this data will be irretrievably lost during the recovery process. Third, DBMS recovery will only deal with tabular state and will thus ignore operator states. For example, in the feed alarm application, the counters are not in stored in tables; therefore their state would be lost in a crash. One straightforward fix would be to force all operator state into tables to use DBMS-style recovery; however, this solution would significantly slow down the application.

The obvious alternative to achieve high availability is to use techniques that rely on Tandem-style process pairs [11]. The basic idea is that, in the case of a crash, the application performs failover to a backup machine, which typically operates as a "hot standby", and keeps going with small delay. This approach eliminates the overhead of logging. As a case in point, StreamBase turns off logging in BerkeleyDB.

Unlike traditional data-processing applications that require precise recovery for correctness, many stream-processing applications can tolerate and benefit from weaker notions of recovery. In other words, failover does not always need to be "perfect". Consider monitoring applications that operate on data streams whose values are periodically refreshed. Such applications can often tolerate tuple losses when a failure occurs, as long as such interruptions are short. Similarly, if one loses a couple of ticks in the feed alarm application during failover, the correctness would probably still be preserved. In contrast, applications that trigger alerts when certain combinations of events happen, require that no tuples be lost, but may tolerate temporary duplication. For example, a patient monitoring application may be able to tolerate duplicate tuples (``heart rate is 79'') but not lost tuples (``heart rate has changed to zero''). Of course, there will always be a class of applications that require strong, precise recovery guarantees. A financial application that performs portfolio management based on individual stock transactions falls into this category.

As a result, there is an opportunity to devise simplified and low overhead failover schemes, when weaker correctness notions are sufficient. A collection of detailed options on how to achieve high availability in a streaming world has recently been explored [17].

4.5 Synchronization

Many stream-based applications rely on shared data and computation. Shared data is typically contained in a table that one query updates and another one reads. For example, the Linear Road application requires that vehicle-position data be used to update statistics on highway usage, which in turn are read to determine tolls for each segment on the highway. Thus, there is a basic need to provide isolation between messages.

Traditional DBMSs use ACID transactions to provide isolation (among others things) between concurrent transactions submitted by multiple users. In streaming systems, which are not multi-user, such isolation can be effectively achieved through simple critical sections, which can be implemented through light-weight semaphores. Since full-fledged transactions are not required, there is no need to use heavy-weight locking-based mechanisms anymore.

In summary, ACID properties are not required in most stream processing applications, and simpler, specialized performance constructs can be used to advantage.

5. One size fits all?

The previous section has indicated a collection of architectural issues that result in significant differences in performance between specialized stream processing engines and traditional DBMSs. These design choices result in a big difference between the internals of the two engines. In fact, the run-time code in StreamBase looks nothing like a traditional DBMS run-time. The net result is vastly better performance on a class of real-time applications. These considerations will lead to a separate code line for stream processing, of course assuming that the market is large enough to facilitate this scenario.

In the rest of the section, we outline several other markets for which specialized database engines may be viable.

5.1 Data warehouses

The architectural differences between OLTP and warehouse database systems discussed in Section 2 are just the tip of the iceberg, and additional differences will occur over time. We now focus on probably the biggest architectural difference, which is to store the data by column, rather than by row.

All major DBMS vendors implement record-oriented storage systems, where the attributes of a record are placed contiguously in storage. Using this "row-store" architecture, a single disk write is all that is required to push all of the attributes of a single record out to disk. Hence, such a system is "write-optimized" because high performance on record writes is easily achievable. It is easy to see that write-optimized systems are especially effective on OLTP-style applications, the primary reason why most commercial DBMSs employ this architecture.

In contrast, warehouse systems need to be "read-optimized" as most workload consists of ad-hoc queries that touch large amounts of historical data. In such systems, a "column-store" model where the values for all of the rows of a single attribute are stored contiguously is drastically more efficient (as demonstrated by Sybase IQ [6], Addamark [1], and KDB [2]).

With a column-store architecture, a DBMS need only read the attributes required for processing a given query, and can avoid bringing into memory any other irrelevant attributes. Given that records with hundreds of attributes (with many null values) are becoming increasingly common, this approach results in a sizeable performance advantage for warehouse workloads where typical queries involve aggregates that are computed on a small number of attributes over large data sets. The first author of this paper is engaged in a research project to explore the performance benefits of a column-store system.

5.2 Sensor networks

It is not practical to run a traditional DBMS in the processing nodes that manage sensors in a sensor network [21, 24]. These emerging platforms of device networks are currently being explored for applications such as environmental and medical monitoring, industrial automation, autonomous robotic teams, and smart homes [16, 19, 26, 28, 29].

In order to realize the full potential of these systems, the components are designed to be wireless, with respect to both communication and energy. In this environment, bandwidth and power become the key resources to be conserved. Furthermore, communication, as opposed to processing or storage access, is the main consumer of energy. Thus, standard DBMS optimization tactics do not apply and need to be critically rethought. Furthermore, transactional capabilities seem to be irrelevant in this domain.

In general, there is a need to design flexible, light-weight database abstractions (such as TinyDB [18]) that are optimized for data movement as opposed to data storage.

5.3 Text search

None of the current text search engines use DBMS technology for storage, even though they deal with massive, ever-increasing data sets. For instance, Google built its own storage system (called GFS [15]) that outperforms conventional DBMS technology (as well as file system technology) for some of the reasons discussed in Section 4.

A typical search engine workload [12, 15] consists of a combination of inbound streaming data (coming from web crawlers), which needs to be cleaned and incorporated into the existing search index, and ad hoc look-up operations on the existing index. In particular, the write operations are mostly append-only and read operations sequential. Concurrent writes (i.e., appends) to the same file are necessary for good performance. Finally, the large number of storage machines, made up of commodity parts, ensure that failure is the norm rather than the exception. Hence, high availability is a key design consideration and can only be achieved through fast recovery and replication.

Clearly, these application characteristics are much different from those of conventional business-processing applications. As a result, even though some DBMSs has built-in text search capabilities, they fall short of meeting the performance and availability requirements of this domain: they are simply too heavy-weight and inflexible.

5.4 Scientific databases

Massive amounts of data are continuously being gathered from the real-world by sensors of various types, attached to devices such as satellites and microscopes, or are generated artificially by high-resolution scientific and engineering simulations.

The analysis of such data sets is the key to better understanding physical phenomena and is becoming increasingly commonplace in many scientific research domains. Efficient analysis and querying of these vast databases require highly-efficient multi-dimensional indexing structures and application-specific aggregation techniques. In addition, the need for efficient data archiving, staging, lineage, and error propagation techniques may create a need for yet another specialized engine in this important domain.

5.5 XML databases

Semi-structured data is everywhere. Unfortunately, such data does not immediately fit into the relational model. There is a heated ongoing debate regarding how to best store and manipulate XML data. Even though some believe that relational DBMSs (with proper extensions) are the way to go, others would argue that a specialized engine is needed to store and process this data format.

6. A Comment on Factoring

Most stream-based applications require three basic services:

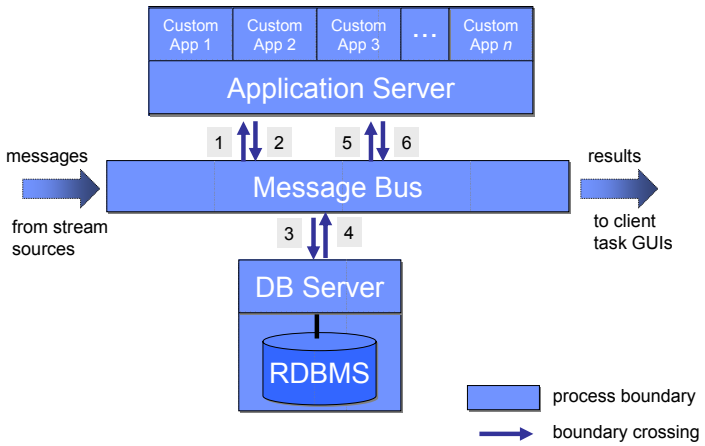

Figure 7: A multi-tier stream processing architecture

- *Message transport*: In many stream applications, there is a need to transport data efficiently and reliably among multiple distributed machines. The reasons for these are threefold. First, data sources and destinations are typically geographically dispersed. Second, high performance and availability requirements dictate the use of multiple cooperating server machines. Third, virtually all big enterprise systems consist of a complicated network of business applications running on a large number of machines, in which an SPE is embedded. Thus, the input and outputs messages to the SPE need to be properly routed from and to the appropriate external applications.
- *Storage of state*: As discussed in Section 4.3, in all but the most simplistic applications, there is a need to store state, typically in the form of read-only reference and historical tables, and read-write translation (e.g., hash) tables.
- *Execution of application logic*: Many streaming applications demand domain-specific message processing to be interspersed with query activity. In general, it is neither possible nor practical to represent such application logic using only the built-in query primitives (e.g., think legacy code).

A traditional design for a stream-processing application spreads the entire application logic across three diverse systems: (1) *a messaging system* (such as MQSeries, WebMethods, or Tibco) to reliably connect the component systems, typically using a publish/subscribe paradigm; (2) *a DBMS* (such as DB2 or Oracle) to provide persistence for state information; and (3) *an application server* (such as WebSphere or WebLogic) to provide application services to a set of custom-coded programs. Such a three-tier configuration is illustrated in Figure 7.

Unfortunately, such a design that spreads required functionality over three heavyweight pieces of system software will not perform well. For example, every message that requires state lookup and application services will entail multiple process switches between these different services.

In order to illustrate this per message overhead, we trace the steps taken when processing a message. An incoming message is first picked up by the bus and then forwarded to the custom application code (step 1), which cleans up and then processes the message. If the message needs to be correlated with historical data or requires access to persistent data, then a request is sent to the DB server (steps 2-3), which accesses the DBMS. The response follows the reverse path to the application code (steps 4-5). Finally, the outcome of the processed message is forwarded to the client task GUI (step 6). Overall, there are six "boundary crossings" for processing a single message. In addition to the obvious context switches incurred, messages also need to transformed on-the-fly, by the appropriate adapters, to and from the native formats of the systems, each time they are picked up from and passed on to the message bus. The result is a very low useful work to overhead ratio. Even if there is some batching of messages, the overhead will be high and limit achievable performance.

To avoid such a performance hit, a stream processing engine must provide all three services in a single piece of system software that executes as one multi-threaded process on each machine that it runs. Hence, an SPE must have elements of a DBMS, an application server, and a messaging system. In effect, an SPE should provide specialized capabilities from all three kinds of software "under one roof".

This observation raises the question of whether the current factoring of system software into components (e.g., application server, DBMS, Extract-Transform-Load system, message bus, file system, web server, etc.) is actually an optimal one. After all, this particular decomposition arose partly as a historical artifact and partly from marketing happenstance. It seems like other factoring of systems services seems equally plausible, and

it should not be surprising to see considerable evolution of component definition and factoring off into the future.

7. Concluding Remarks

In summary, there may be a substantial number of domain-specific database engines with differing capabilities off into the future. We are reminded of the curse "may you live in interesting times". We believe that the DBMS market is entering a period of very interesting times. There are a variety of existing and newly-emerging applications that can benefit from data management and processing principles and techniques. At the same time, these applications are very much different from business data processing and from each other — there seems to be no obvious way to support them with a single code line. The "one size fits all" theme is unlikely to successfully continue under these circumstances.

References

[1] Addamark Scalable Log Server. http://www.addamark.com/products/sls.htm.

[2] Kx systems. http://www.kx.com/.

[3] Lojack.com, 2004. http://www.lojack.com/.

[4] Sleepycat software. http://www.sleepycat.com/.

[5] StreamBase Inc. http://www.streambase.com/.

[6] Sybase IQ. http://www.sybase.com/products/databaseservers/sybaseiq.

[7] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Zing, R.Yan, and S. Zdonik. Aurora: A Data Stream Management System (demo description). In *Proceedings of the 2003 ACM SIGMOD Conference on Management of Data*, San Diego, CA, 2003.

[8] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 2003.

[9] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Benchmark for Stream Data Management Systems. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, Toronto, CA, 2004.

[10] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. Wade, and V. Watson. System R: A Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1976.

[11] J. Barlett, J. Gray, and B. Horst. Fault tolerance in Tandem computer systems. Tandem Computers Technical Report 86.2., 1986.

[12] E. Brewer, "Combining systems and databases: a search engine retrospective," in *Readings in Database Systems*, M. Stonebraker and J. Hellerstein, Eds., 4 ed, 2004.

[13] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002.

[14] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the 1st CIDR Conference*, Asilomar, CA, 2003.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP)*, Bolton Landing, NY, USA, 2003.

[16] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh. An Energy-Efficient Surveillance System Using Wireless Sensor Networks. In *MobiSys'04*, 2004.

[17] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proceedings of the International Conference on Data Engineering*, Tokyo, Japan, 2004.

[18] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of SIGMOD*, San Diego, CA, 2003.

[19] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton. CodeBlue: An Ad Hoc Sensor Network Infrastructure for Emergency Medical Care. In *WAMES'04*, 2004.

[20] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation and in a Data Stream Management System. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, 2003.

[21] G. Pottie and W. Kaiser. Wireless Integrated Network Sensors. *Communications of the ACM*.

[22] K. Rothermel and C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In *Proc. 15th International Conference on Very Large Data Bases (VLDB)*, Amsterdam, Holland, 1989.

[23] L. A. Rowe and K. A. Shoens. Data abstraction, views and updates in RIGEL. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data (SIGMOD)*, Boston, Massachusetts, 1979.

[24] P. Saffo. Sensors: The Next Wave of Information. *Communications of the ACM*.

[25] J. W. Schmidth. Some High-Level Language Constructs for Data of Type Relation. *Transactions on Database Systems*, 2(247-261, 1977.

[26] L. Schwiebert, S. Gupta, and J. Weinmann. Research Challenges in Wireless Networks of Biomedical Sensors. In *Mobicom'01*, 2001.

[27] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The Design and Implementation of INGRES. *ACM Trans. Database Systems*, 1(3):189-222, 1976.

[28] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a Sensor Network Expedition. In *EWSN'04*, 2004.

[29] C. S. Ting Liu, Pei Zhang and Margaret Martonosi. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *MobiSys'04*, 2004.