One Size Fits All? – Part 2: Benchmarking Results

Michael Stonebraker[1], Chuck Bear[2], Uğur Çetintemel[3], Mitch Cherniack[4], Tingjian Ge[3]
Nabil Hachem, Stavros Harizopoulos[1], John Lifter[5], Jennie Rogers[3], and Stan Zdonik[3]

M.I.T.[1], Vertica Inc.[2], Brown University[3], Brandeis University[4], Streambase Inc.[5]

ABSTRACT

Two years ago, some of us wrote a paper predicting the demise of "One Size Fits All (OSFA)" [Sto05a]. In that paper, we examined the stream processing and data warehouse markets and gave reasons for a substantial performance advantage to specialized architectures in both markets. Herein, we make three additional contributions. First, we present reasons why the same performance advantage is enjoyed by specialized implementations in the text processing market. Second, the major contribution of the paper is to show "apples to apples" performance numbers between commercial implementations of specialized architectures and relational DBMSs in both stream processing and data warehouses. Finally, we also show comparison numbers between an academic prototype of a specialized architecture for scientific and intelligence applications, a relational DBMS, and a widely used mathematical computation tool. In summary, there appear to be at least four markets where specialized architectures enjoy an overwhelming performance advantage.

1. The History of the OSFA Architecture

Relational Database Management System (RDBMS) technology dates from the work of System R [Ast76] and Ingres [Sto76]. At the time (1970s), the architects of these systems were focused on proving that relational technology was superior to hierarchical and network systems on the tasks that were then common, namely business data processing. Hence, architectural decisions in these early prototypes were focused on transaction processing. Essentially all commercial RDBMSs are direct descendents of these early systems and share their common architecture (such as row store representation, B-tree indexing, modest disk block size, and tuple-oriented execution).

Over the years, the large RDBMS vendors have enhanced this original architecture in a number of ways, which include:

- **Multi-processor configurations**. Originally designed for shared-memory multi-processors, commercial RDBMS systems have been extended to support either shared disk systems (disk clusters), shared nothing systems (blades), or both.
- **XML**. Recently, several RDBMSs have been extended to support either SQL or XQuery on either tables or data represented in XMLSchema.
- **Data Warehouses**. Several commercial systems have been extended with features designed to make business intelligence (warehouse) query workloads perform better. These include techniques such as data compression, materialized view, index-only tables, and join indexes.

The main purpose of these various enhancements is to continue to sell a single code line supporting all DBMS needs. The reasons for this "one size fits all" (OSFA) strategy include the following:

- **Engineering costs**. Multiple code lines increases engineering effort linearly with the number of code lines. Moreover, multiple code lines must often be kept synchronized, resulting in additional engineering effort.
- **Sales costs**. RDBMS salesmen must be taught which code line to sell in which circumstance. Since the issues get complex quickly (and salespeople are known not to be rocket scientists), this is a daunting task.
- **Marketing costs**. Multiple code lines must be carefully positioned in the marketplace. This is often a difficult challenge. It is much simpler to position OSFA, i.e., "I am the guy with the hammer, and everything is a nail".

A single code line will succeed whenever the intended customer base is reasonably uniform in their feature and query requirements. One can easily argue this uniformity for business data processing. However, in the last quarter

3rd *Biennial Conference on Innovative Data Systems Research (CIDR)*
January 7-10, 2007, Asilomar, California, USA.

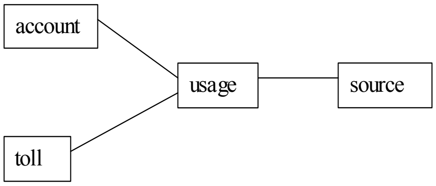

Figure 1. Telco Schema

	Vertica	*Appliance*
Query 1	2.06	300
Query 2	2.20	300
Query 3	0.09	300
Query 4	5.24	300
Query 5	2.88	300

Figure 2. Query Running Times (seconds)

century, a collection of new markets with new requirements has arisen. In addition, the relentless advance of technology has a tendency to change the optimization tactics from time to time.

When either of these occurs, there is a possibility that a new code line with a different architecture will dramatically outperform the traditional one. Inevitably, this is the result of a different storage architecture that has inherent advantages relative to the OSFA one. For the purpose of this paper, we define "dramatically outperform" to mean *at least* a factor of 10 advantage on the same (or comparable) hardware. For example, a factor of 10 is the difference between response time of one minute and response time of 6 seconds. Similarly, it is the difference between an $800 PC with two CPUs and a blade farm with 20 processors. Whenever such a performance difference occurs, customers who care about performance (i.e., ones that are "in pain") will be inclined to try the new architecture. Although one can argue about whether a factor of 10 is too high a fence for a new architecture to clear, the number is clearly not a factor of 2 or 3. In the latter case, one merely waits a year or two for the next hardware advance or increases the hardware budget. A factor of 10, in contrast, makes such tactics unworkable.

The premise of this paper is that there are at least four markets where this factor of 10 (or higher) threshold currently exists. In the next four sections, we detail the reasons for our claims, which are based on benchmarking or reports of benchmarking results by others. In the last section of this paper, we speculate on a few ways that the commercial DBMS market could unfold off into the future.

2. Text Databases – A Factor of 10

It is a significant disappointment that text storage and retrieval engines do not use RDBMSs, a comment repeated at most DBMS conferences. In fact, this market does not use any DBMS, preferring to build directly on top of a file system storage layer. Early warning of this "roll your own" phenomenon came to one of us from the founder of Inktomi (Eric Brewer) in the mid 1990s. He tried using a commercial RDBMS in an early version of their product, but quickly gave up when he realized that Inktomi ran exactly one query, a three way join with constants for the search terms in the user query. This single query could be easily hard coded and ran about 100 times faster than the same query in an RDBMS.

There are a myriad of reasons for this performance delta. These include (i) the lack of need for locking or transactions, data types other than text, and repeatable or even complete answers, and (ii) the need for horizontal data partitioning, application-specific compression, and variable length lists.

In a retrospective, Brewer [Bre04] explored these reasons in some detail. Moreover, all subsequent search engines (e.g., Google, Lycos, etc.) have come to the same conclusion and have built proprietary text engines. Moreover, Google has built a complete system software stack including a file system (GFS [Ghe03]), a special DBMS (Bigtable [Cha06]), and pertinent parallel data processing abstractions (MapReduce [Dea04] and Sawzall [Pik05]). Bigtable is being deployed for a myriad of internal storage uses.

At this point, it is likely that one of the search companies will expose their internal storage system for customer data, either as an appliance on the customer's premise (along the lines of the current Google appliance) or as a service. When this happens, there will be one or more prominent non-RDBMS architectures used to store customer data.

3. Data Warehouses – Another Factor of 10

It is estimated that data warehouses form 1/3 of the RDBMS market in 2005 [Gar06, Ola06]. Right now, the data warehouse market is dominated by RDBMS vendors selling systems that use the traditional row-oriented architecture. C-Store [Sto05b] and Monet [Bon04] advocated the use of a column store for data warehouse applications and [Har06] gave some preliminary performance numbers. In this section, we present additional evidence, namely two specific performance studies using the now-released code line of Vertica [Ver06], a complete column-oriented DBMS along the lines of C-Store, which validate the column store performance claims.

3.1 Telco Call Details

Most (if not all) data warehouse applications use a star or snowflake schema, and the schema for this application is shown in Figure 1. This schema is in production use by a firm that specializes in business analysis of Telco call

```
SELECT account.account_number,
       sum (usage.toll_airtime),
       sum (usage.toll_price)
FROM   usage, toll, source, account
WHERE  usage.toll_id = toll.toll_id
  AND  usage.source_id = source.source_id
  AND  usage.account_id = account.account_id
  AND  toll.type_ind in (‘AE’. ‘AA’)
  AND  usage.toll_price > 0
  AND  source.type != ‘CIBER’
  AND  toll.rating_method = ‘IS’
  AND  usage.invoice_date = 20051013
GROUP BY  account.account_number
```

Figure 3. Query 2

detail information. Here, the central fact table (usage) has a record per call with a variety of call detail data; the account table contains the phone numbers which are the billing entities; source contains the network the call detail came from; and toll contains billing information.

Besides their schema, this firm gave Vertica 600 Gbytes of actual data and a suite of example queries they use in their day-to-day affairs. Moreover, they also gave Vertica approximate running times for their current solution, a 28 blade appliance from one of the well-known DBMS appliance vendors, which lists for about $300,000 and implements a traditional row-store architecture. Figure 2 shows these approximate running times, as well as the performance of Vertica on a dual core dual CPU Opteron computer, which lists for about $2500.

Figure 3 shows the SQL for Query 2 of this benchmark suite defined over the star schema shown in Figure 1. This query is typical of data warehouse queries in that it first filters using predicates over columns of the fact table or a dimension table, and then groups the restricted fact table over some attribute and aggregates. On this query, a column store outperforms a row store by a factor of 47 with 1/7 the number of CPUs and two orders of magnitude less hardware cost. There are several reasons for this startling performance difference, but three stand out.

First, the usage table contains a myriad of details about calls, including call forwarding information, the networks the call traversed, call length and drop information, etc. In all, there are more than 200 columns. While the wisdom of such a “fat” table can be debated by schema designers, it should be noted that the customer uses many different fields in various ad-hoc queries. Hence, decomposing this fact table into multiple tables would introduce joins, which might slow performance. Also, in point of fact, the fact table has been pre-joined with appropriate columns in dimension tables to eliminate run-time joins, a common tactic to improve performance in warehouse environments, resulting in a materialized view with 212 columns. Note further that query 2 reads 7 columns from the 212. Hence, a column store will read exactly the 7 columns, while a row store will read all 212—a striking difference of nearly two orders of magnitude in byte movement from the disk.

The second consideration is compression. As noted in a companion paper [Aba06], compression is usually more effective in a column store than a row store. Not only are all objects in a disk block of the same data type, but additional compression options, such as delta encoding and run-length encoding, are possible. In several benchmark studies, Vertica typically has a compression ratio of a factor of 10, better by nearly a factor of 3 than the compression possible in competing row stores.

The third reason is sorting and indexing, which are used by Vertica, but not by the appliance. This explains the constant query time of the appliance, whereas Vertica keeps the data in some sort order, which can restrict running times for some queries.

It is now straightforward to explain Figure 2. The competing row store did not use compression or indexing. Hence, query times for the appliance are the time to read 600 Gbytes off the disk, an impressive 70 Mbytes/sec per processor-disk pair. In contrast, Vertica stored less than 60 Gbytes and actually read about 2 Gbytes. A factor of 300 less I/O is guaranteed to generate a dramatic performance improvement!

Although this customer had a very fat fact table, which obviously skewed the performance comparison, similar, though likely less dramatic, results have been observed in a variety of other studies. Our next example uses a “skinny” fact table.

3.2 Simplified TPC-H

The well known benchmark, TPC-H, is used by many vendors to claim superiority in data warehouse performance. This benchmark is cleverly constructed to avoid using a snowflake schema and to render materialized views unproductive. In interviewing about two dozen CIOs, the authors have never seen a warehouse that did not use a snowflake schema. Hence, Pat O’Neil simplified the TPC-H schema to be a snowflake and defined variants of 12 TPC-H queries on this schema [One06]. The schema is shown in Figure 4 and a few of the 12 queries in Figure 5. Lastly, the running time of the 12 queries on a $2500 Opteron computer with 4 cores is shown in Figure 6 for two engines, the Vertica column store and a popular row store.

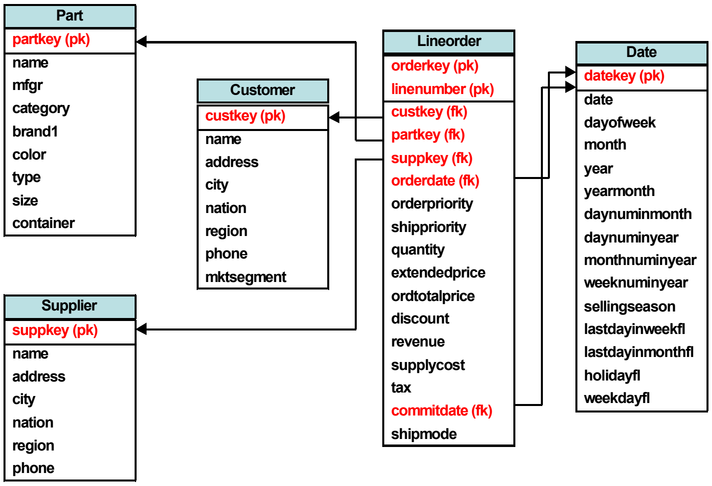

Figure 4. The Schema for Simplified TPC-H

Both systems used compression and horizontal partitioning. Moreover, the row store was optimized by a "4 star wizard" DBA who tunes this vendor's product as his profession. In Figure 6, we report on two physical schemas. The first one is called "low space" and uses very modest redundancy, while the second, called "medium space" creates three materialized views so no query need perform a complete scan. Lastly, the benchmark uses scale 100 sizes, as defined in TPC-H, and the raw data occupies approximately 60 Gbytes.

In Figure 6, one can compare the two systems using identical physical schemas. Note that the column store is around 7 times faster in less than half the space. Alternately, one could compare the two systems giving each a space budget. In this case, the column store is a much larger factor faster than the row store. Although these results are less dramatic than the results of Section 3.1, they are similar to the results reported in [Sto05b].

4. Stream Processing – Another Factor of 10

Recently, there has been considerable interest in performing low latency processing of message streams using a high-level tool kit. There are commercial products which use a rule notation (e.g., Apama [Apa06]), as well as ones which use a SQL notation (e.g., StreamBase [Str06] and Coral8 [Cor06]). Although there is some debate over

Q1: Select the total discount given in 1993, for lineitems with <25 quantity, and 1-3% discount	*Q5: Measure total revenue for a certain manufacturer/category/brand of part, from a supplier in a certain geographical area.*	*Q8: Measure the revenue from customers in the same geographical region as their suppliers, further broken down by geography and year.*
SELECT SUM (lo_extendedprice*lo_discount) **AS** revenue **FROM** lineorder, dwdate **WHERE** lo_orderdate = d_datekey **AND** d_year = 1993 **AND** lo_discount between 1 and 3 **AND** lo_quantity < 25;	**SELECT SUM** (lo_revenue), d_year, p_brand1 **FROM** lineorder, dwdate, part, supplier **WHERE** lo_orderdate = d_datekey **AND** lo_partkey = p_partkey **AND** lo_suppkey = s_suppkey **AND** p_brand1 between 'MFGR#2221' and 'MFGR#2228' **AND** s_region = 'ASIA' **GROUP BY** d_year, p_brand1 **ORDER BY** d_year, p_brand1;	**SELECT** c_city, s_city, d_year, sum(lo_revenue) as revenue **FROM** customer, lineorder, supplier, dwdate **WHERE** lo_custkey = c_custkey **AND** lo_suppkey = s_suppkey **AND** lo_orderdate = d_datekey **AND** c_nation = 'UNITED STATES' **AND** s_nation = 'UNITED STATES' **AND** d_year between 1992 and 1997 **GROUP BY** c_city, s_city, d_year **ORDER BY** d_year asc, revenue desc;

Figure 5. Samples from the Query Set of 12

	Row Store Low Space	*Column Store Low Space*	*Row Store High Space*	*Column Store High Space*
Query 1	32.0	3.4	3.9	1.1
Query 2	31.6	4.1	1.6	0.3
Query 3	30.8	2.8	0.9	0.2
Query 4	29.3	3.4	7.2	0.6
Query 5	26.1	3.2	2.1	0.2
Query 6	22.2	3.0	0.7	0.1
Query 7	60.9	1.7	15.6	2.4
Query 8	4.1	3.2	2.5	0.3
Query 9	3.7	3.0	2.0	0.2
Query 10	24.1	1.1	11.4	1.8
Query 11	5.1	0.2	6.3	0.3
Query 12	0.7	0.2	1.0	0.2
Weighted Average	13.6	1.8	2.9	0.4
Space Required	60.3	36.3	76.7	40.2

Figure 6. Running Time (seconds) and Space Requirements (GBytes)

which paradigm will win, it appears that a SQL notation has one big advantage, namely that most real-time stream processing problems have an embedded requirement to store and access substantial amounts of state. Since SQL is the universal paradigm for stored data, it is natural to use an extended version of SQL for the required mixing of real time and historical data.

The current commercial systems are descendents of academic prototypes, such as Aurora [Aba03] and STREAM [Mot03]. At this point, there is substantial marketplace experience with such commercial systems, and their performance relative to the alternatives, namely using custom code or a relational DBMS. In this section, we also discuss two benchmarks comparing a specialized engine with a relational DBMS. Although Linear Road [Ara04] would be a natural choice, we chose instead to use two scenarios that came from real customers.

4.1 Split Adjusted Price

This is a calculation that is often done in Wall Street applications. There is an incoming feed of tick data:

Ticks (symbol = C6,
time = double,
volume = integer,
price = double)

which could come from one of the popular data providers or from a direct connection to an exchange. In addition, there is a second feed:

Splits (symbol = C6,
time = double,
split_factor = float)

The second feed records the times at which stock splits take place, giving the split factor; i.e., a 2 for 1 split would have a split factor of 2 and a 1 for 2 (reverse) split would have a split factor of 0.5.

The goal of the computation is to produce the split adjusted price for the **Ticks** feed. Since a given symbol may split more than once, it is necessary to maintain a running composite of the total split factor seen up to now. This state is maintained in a **Storage** table

Storage (symbol = C6,
total_split = float)

The following StreamSQL statements indicate the required processing on the stored table, **Storage,** and the two feeds **Ticks** and **Splits**.

```
UPDATE Storage
FROM Splits S
SET (total_split = total_splits *
       S.split_factor)
WHERE S.symbol = Storage.Symbol

SELECT T.symbol,
       price = T.price * S.factor,
       T.volume, T.time
FROM Ticks T, Storage S
WHERE S.symbol = T.symbol
```

The above code was run on StreamBase on a $1000 system (2.8GHz Pentium D 820, 3GB RAM, and 4x200G SATAII drives) and yielded a throughput of 333,000 messages per second. The RDBMS logic to accomplish the same task is a bit tricky. One option is to create a stored table Storage and then perform the rest of the application in a stored procedure. This approach yielded 12,640 messages per second. A second approach is to insert both feeds into the DBMS and then use triggers to perform the required processing. This approach has the characteristic of more fully utilizing DBMS functionality, but slows performance dramatically.

To show the RDBMS in its best light, we also implemented a third alternative, namely allocating the Splits table as an array inside the stored procedure. To a first approximation this is a solution which codes the entire application as a stored procedure, using no DBMS facilities. Obviously, one would prefer to do the application with custom logic totally external to the DBMS, in preference to this option. However, we report this result in the category of "if you stand on your head, this is as good as it gets", and the result was 38,000 messages/second.

4.2 Forward First Arriver

It is common for Wall Street firms to subscribe to multiple stock ticker feeds, such as Reuters, Comstock, or Infodyne. An equally common application is to forward the first arriving tick from whichever of the feeds has least latency at that moment. Late duplicates must be discarded, thereby creating a single virtual feed with the first arriving tick information.

Obviously, one must forward the first arriver without waiting for the late duplicates. Otherwise, the whole purpose of the application, latency reduction, would be lost. In the interest of brevity, the StreamSQL for this application is omitted. However, the StreamBase implementation on the $1000 Pentium PC yielded performance of 281,000 messages per second. In contrast, the RDBMS logic requires one to construct a table of recent ticks and then to check this table to see if an incoming tick is a late duplicate. If so, it is discarded; otherwise it is forwarded and inserted into the table. The best performance our "4-star" wizard could coax out of the RDBMS was 11,790 messages per second using a table for the recent ticks and 51,700 messages/sec using an array inside the stored procedure.

In [Sto05a], we discussed the reasons why stream processing engines outperform RDBMSs in this application. Here, we briefly review the reasons:

- StreamSQL engines do not run in client-server mode. Hence, there are no process switches. In contrast, RDBMS engines are built not to trust the application, and must run it in a separate address space from the application.
- Most StreamSQL engines do processing exclusively in virtual memory. There is no concept of reliably storing the data on disk, which is just a source of extra overhead.
- StreamSQL engines support time windows natively, and do not need to simulate them using conventional SQL notions.
- It pays to compile predicates to machine code for maximum performance, in contrast to RDBMSs, which typically compile operations to an intermediate form for easier maintenance.
- RDBMSs are optimized for joining a million record source table to a million record target table. In contrast, StreamSQL engines are optimized for processing a single message (tuple) through a collection of operations with minimum latency. Hence, avoiding queues between operations is a good idea, whereas in RDBMSs such queues are omni-present.

In summary, an in-process engine using main memory storage and optimized for single message operations will always greatly outperform a disk-based engine that uses out-of-process storage and that is optimized for set processing.

5. Scientific and Intelligence Applications – Another Factor of 10

Scientific and intelligence users have historically shunned commercial data base products, preferring to use customized solutions, such as HDF-5 [Hdf06], MatLab [Mat06], and NetCDF [Net06]. There are a myriad of reasons for this. In this section, we briefly review the lessons learned from Project Sequoia in the mid 1990s [Seq93]. After that, we present results from comparing a specialized prototype, ASAP ([A]rray [S]treaming [A]nd [P]rocessing), with a popular commercial RDBMS.

5.1 Project Sequoia

The DEC-sponsored Sequoia project [Seq93] attempted to support scientific DBMS users (specifically the Earth Science research group at UC Santa Barbara under the direction of Jeff Dozier and the climate modeling group at UCLA under the direction of Roberto Mechosa) by applying POSTGRES to their problems. The result was unsuccessful. The major reason was that POSTGRES had no support for large multi-dimensional arrays, and the vast preponderance of objects from both groups was array data. Simulating arrays on top of POSTGRES tables was inefficient and inflexible. A second lesser reason was that POSTGRES had no built-in support for meta-data management. Santa Barbara scientists required information on the processing steps that had been applied to each of their data sets; mostly the data cleansing and data reduction algorithms which "cooked" raw satellite imagery into usable information. In addition, essentially all scientific data that results from real world observations is fundamentally uncertain, and error metrics are required for

	Number of Dimensions			
	3	*4*	*5*	*6*
Speedup over Matlab	82.9	16.44	9.6	10.1
Speedup over RDBMS	102.5	119.1	114	119

Figure 7. ASAP Speedup numbers for
Dot Product (Low Stride case)

such data sets. POSTGRES had no capabilities to store and update either data uncertainty or lineage.

Revisiting this problem area a decade later strongly suggests the need for a different approach than Object-Relational DBMSs. To satisfy the needs of this community, we are building a new DBMS, called ASAP, which uses multi-dimensional arrays as the basic storage and processing object. We have enough of ASAP running to perform the following two benchmarks. Although artificial, they appear to capture some of what scientific users want to do.

5.2 Array Benchmarks

5.2.1 Dot Product

In two dimensions, dot product is defined as follows. Given two arrays, A[I, J] and B[I, J], we compute the dot product over the shared dimension J as:

C [I], where C[i] = sum over j of A[i, j] * B [i, j]

Notice that this is a straightforward generalization of the standard vector dot product for arrays: we compute the dot products of all vector pairs that we derive by iterating through all the values of the remaining dimensions.

The benchmark is to compute the dot product of two integer arrays, A and B, with a number of dimensions varying between 3 and 6. The benchmark was first run on ASAP and Matlab, both of which support arrays as a native data type. In this experiment, each array has 250M elements (2GB double precision raw data), which were distributed uniformly across all dimensions. We compute the dot product for each possible shared dimension and compute the average running time of this collection of matrix computations. The results are obtained on a 2 GHz, 64-bit Athlon machine with 1GB RAM.

The same operation was also run on ASAP and a popular commercial RDBMS, in which each array is represented as a table with attributes: dimension-1, …, dimension-n, and value. In this case, each array consists of 25M elements (100MB single precision raw data) and a 3.2 GHz Pentium machine with 1GB RAM was used. For this dataset, the RDBMS uses 502MB-680MB per array, depending on the number of dimensions, as it has to store the dimensions as well.

	Number of Dimensions			
	3	*4*	*5*	*6*
Speedup over Matlab	2.1	3.16	9.62	9.7
Speedup over RDBMS	270.9	775.1	798	738

Figure 8. ASAP Speedup numbers for
Matrix Multiplication (Low Stride case)

Finally, we examine two scenarios: (i) a “low stride” scenario, where all elements of a vector that is operated on are stored contiguously (as much as possible); and (ii) a “high stride” scenario where the elements of an operand vector are rather dispersed throughout the array. The former scenario models a simple, almost serial array access pattern, whereas the latter models a more complicated one.

Figure 7 shows the speedup that ASAP achieves over Matlab and the RDBMS for these experiments for the low stride case. ASAP is always advantageous and achieves a speedup of up to 83 over Matlab. With increasing number of dimensions, the operand vector sizes decrease, which leads to a decrease in the speedup values as Matlab utilizes the memory better with smaller operands. In the high stride cases (not shown) as well as with datasets that are much larger than main memory, Matlab heavily relies on the virtual memory and basically comes to a halt.

The comparison with the RDBMS consistently yields a speedup of 100 or more (similar results are obtained for the high-stride cases).

This dot product computation was initially chosen because it is a modest computation, and one can craft a single pass algorithm to perform this function. Hence, it is relatively modest in both I/O and CPU requirements. The next section extends the benchmark to a more complex array operation.

5.2.2 Matrix Multiplication

In two dimensions, the matrix multiplication of two arrays, A[I, J] and B[J, L], is defined as:

C [I, L], where C[i, l] = sum over j of A[i, j] * B [j, l].

The benchmark is to compute the product of two integer arrays, A and B, with a number of dimensions varying between 3 and 6. For a given set of two shared dimensions, we compute the multiplications of all matrix pairs that we derive by iterating through all the values of the remaining set of dimensions. Note that this operation is *N* times more intensive than the dot product, where *N* is the average size of a dimension.

This calculation was performed using ASAP as well as Matlab and the same RDBMS. Figure 8 tabulates the speedup results for the low stride case on the configurations described earlier for the dot product.

We can observe even more dramatic improvements (~800x) over the RDBMS as matrix multiplication is not amenable to a single pass algorithm. On the other hand, Matlab does a much better job here, as matrix multiplication is a built-in, highly optimized operation.

Even so, ASAP beats Matlab for all dimensions. In the high-stride case and with larger datasets, however, the performance of Matlab deteriorates very fast due to heavy swapping.

5.2.3 A Note on the RDBMS Implementation

Both array benchmarks are implemented inside the RDBMS as a single SQL query that uses a group-by aggregation to compute the sum of the individual multiplications. Letting the RDBMS handle all multiplications incurs a certain overhead in that all of the dimension information needs to be copied over throughout the group-by computations.

Perhaps a better approach would be to use the RDBMS to first correctly sort all pairs of vectors (or matrices) and then implement both types of computations in user space. Still, we expect such an approach to be significantly slower than ASAP since ASAP does not to perform any type of sorting. Furthermore, as we discuss next, ASAP has to read only a fraction of the bytes read by the RDBMS.

5.3 Reasons for a Large Performance Difference

We now explore the reasons for the performance difference on the two benchmarks. In addition, we speculate about the possible performance differential on a third benchmark that has been omitted because of lack of time. This third benchmark is to process a sequence of arrays, which might come from a surveillance system such as a video camera, a radar system, or satellite observation system, looking for a specific predefined pattern, P [I, J]. P would be an image of interest, such as a particular kind of vehicle. Whenever, a “hit” is detected, it should be reported.

5.3.1 ChunkyStore

The fundamental object in ASAP is a multi-dimensional array, A(*I*, *J*, …, *K*). *I*, *J*, …, *K* are called *indexes* or *dimensions*, and there can be an arbitrary (finite) number of them. ASAP supports a collection of primitive data types (e.g., integer, float, and string) and will likely be extended with a POSTGRES-style abstract data type facility. Hence, any dimension can be of any data type supported by the system. In addition, ASAP assumes that every data type has a POSTGRES-style linear ordering [14]. Hence, the next value of a dimension is always well defined. A dimension may have a *regular stride* (e.g., 1, 2, 3…) or an *irregular stride* (e.g., 1.2, 2.76, 4.3, …). The value of an array at any collection of dimension values is a *tuple* of *attribute values***.** Each attribute is named and has values from a single primitive type and may be NULL. Hence, an array value is basically a relational tuple.

ASAP contains a storage system, *ChunkyStore*, with the following representations for arrays:

- **Dense arrays with regular strides**: Here, the dimensions are not stored, and data elements are packed into storage in a straightforward way. Such arrays are then decomposed into “chunks” which contain all values for a “super-stride” in each of the dimensions. This is similar to the work of Sarawagi [Sar94] in chunking large arrays. Chunks are allocated to large disk blocks (e.g., 64K to 1 Mbytes), such that array elements are easily addressed.
- **Dense arrays with irregular strides**: In this case, there is a dimension index for each dimension that orders the dimension values. This index is then stored in addition to the chunked arrays.
- **Sparse arrays which are regular or irregular**: Only the non-null values are stored, together with their dimension values. Again, the data will be chunked into disk blocks. However, there must be a block level index defining the bounding box, which defines the subarray in each chunk.
- **Hybrid scheme:** If the nulls in an array are highly skewed, it may make sense to store the array as a collection of subarrays, each with a storage representation appropriate to the contents of the subarray.

Notice that the super-stride will be a variable for sparse arrays. Also note that there is automatic indexing for all dimensions, but no indexing for array values. If ASAP users filter frequently on array values, then we will revisit this choice in the future.

Notice that ChunkyStore is highly advantageous in both benchmarks. First, the array dimensions for A and B are not stored. As such, the space consumed by ASAP is a small fraction of that consumed by the RDBMS. Even if the access patterns of the two systems were the same, ASAP would read this small fraction of the bytes read by the RDBMS. Second, the chunking dramatically cuts down on I/O for both benchmarks. In each case, a linear algorithm can be used that will read each chunk just once. In contrast, the RDBMS does not support such chunking and, in the best of circumstances, will require at least an $N * \text{Log } N$ algorithm.

5.3.2 Operators

Besides the obvious relational-style operators (e.g., filter, aggregate), ASAP contains a collection of primitive operations oriented toward scientific computing. These include the following:

- **Conventional array operations**: These include multiplication, addition, Eigenvalue computations, Fourier transforms, etc.
- **Pivot:** This command changes the dimensions of an array, by moving zero or more dimensions to normal values and zero or more normal values to dimensions.

Of course, this may convert a regular array to an irregular one or vice-versa. Also, if the one of the new dimensions is not a key (i.e. its values are not unique), then the result of the pivot operations is not an array and must be disallowed.

- **Regrid:** Arrays may optionally have a *coordinate system*. ASAP supports any number of named coordinate systems. Associated with each co-ordinate system is a collection of functions that map co-ordinate points to other coordinate systems. To convert an array from one coordinate system to another, it must be regridded, requiring a function to map any point in the source co-ordinate system to the target coordinate system. In this way, the lower left hand corner and all strides can be mapped to the target space. However, a cell in the source coordinate system rarely matches a cell in the target coordinate system. Hence, interpolation must be performed. This requires a support function**,** *Intersect* (source_cell, target_cell) that will return the degree of overlap between the cells. An interpolation function can then decide the value to place in each target cell by performing some computation on the intersecting source cells.
- **Concatenate**: Two arrays are *compatible* if they have the same number of dimensions of the same types. Collections of compatible arrays can be formed into groups, by adding an extra dimension, containing the array name using the *Concatenate* command. We note that there is no notion of sets of arrays; rather a set is modeled as an array in one higher dimension.
- **Locate**: This command generalizes join to deal with fuzzy matching and with groups of elements. Locate searches a large array_1 for areas that "fuzzy-match" a smaller array_2, with the matching criteria described by the given function; i.e., the function must return "true" for the area of array_1 that matches array_2. This command is useful for feature extraction, where array_2 is the feature being searched in a data array array_1. In addition, Locate must perform a heuristic search of the large array for instances of the small array. There are any number of search techniques that can be used, a last parameter to the Locate command is the search technique that should be employed. ASAP implements the common ones, and this set can be extended by an interest user.

In both benchmarks, ASAP takes advantage of the matrix multiplication primitive, which is optimized for ChunkyStore. In addition, in the third benchmark, Locate can be used with a heuristic search limit the number of comparisons. Such optimizations are impossible in an RDBMS, which must simulate array operations on top of the conventional relational ones.

5.3.3 Compression

Obviously, dense arrays can be stored without explicit dimensions. In addition, it makes sense in some environments to do further compression. For example, one can delta-encode array values in a chunk, relative to a base value, thereby storing them in a lesser number of bits. Alternatively, one can delta encode arrays values, relative to their predecessor in some dimension. MPEG does exactly this for video in the time dimension, and ASAP will do this in a more general way.

Although such compression is not useful in the two specific benchmarks, it can be profitably used in the third scenario. If the system delta encodes successive images in the time dimension, then Locate can ignore all parts of the image which are identical to the previous one (a common occurrence for stationary surveillance platforms), thereby lowering processing costs dramatically. Such compression is exceptionally difficult to obtain using an RDBMS solution, because the value to be delta encoded may not be in an adjacent tuple, or even on the same disk block.

5.3.4 Seamless Integration of Real-time Cooking and Storage

Most scientists collect raw data from instruments and then execute a workflow of processing steps to "cook" the data, which includes operations to perform such tasks as coordinate system transformations and data cleaning. One of the most important pieces of data lineage is the cooking "recipe", which must be correctly captured in any system.

Moreover, it is important that the cooking system have the same data model as the storage system. Otherwise, data transformations must be present to convert back and forth. Furthermore, if the cooking system becomes overloaded, then partially cooked data must be saved for later processing, which also argues for the same data model in both components. ASAP contains a real-time cooking component, which is a retargeting of the Aurora/Borealis code line to support arrays.

This integration of real-time cooking and storage is not useful in the two benchmarks executed. However, in the third one, we could perform all of the processing required without ever storing the data or performing any task switches. In contrast, an RDBMS must store and then retrieve the data, resulting in substantial extra overhead.

In aggregate, this architecture changes result in the factor of 10 times or more performance improvement, as observed on the dot product and matrix multiplication benchmarks. We would expect even more dramatic improvement if we had time to run the image analysis benchmark.

6. Other ASAP Features and Potential Performance Pitfalls

As mentioned earlier, one of the lessons from the Sequoia project was the need to natively support data uncertainty and lineage in the database. The processing system needs to capture these data and automatically carry them throughout the entire processing pipeline, which can be prohibitively expensive if done naively. In particular, dealing with uncertain, probabilistic data can easily create a performance bottleneck (both CPU and memory), and thus needs special treatment to be practical, especially for the real-time cooking component.

This section describes the tunable approaches that ASAP will use to deal with uncertainty and lineage data. The goal is to achieve higher efficiency at the expense of some (bounded) accuracy, while meeting the needs of a large fraction of scientific applications.

6.1 Probabilistic Treatment of Data

Essentially all scientific data that results from real-world observations is fundamentally uncertain. Previous work (e.g., [Bar92], [Wid05]) addressed inaccurate or probabilistic data in traditional databases. ASAP focuses on uncertainty in multidimensional array processing. In this context, uncertainty can arise in several ways:

- **Value uncertainty**: An array value invariably has measurement error, which results in the actual value being uncertain. This is the typical probabilistic data support in databases.
- **Position (dimension value) uncertainty**: In certain cases, the very position of the measurement is imprecise, as opposed to the obtained data value. Accordingly, the dimension values in the array are uncertain.
- **Result uncertainty of functions or predicates**: Some functions or predicates, even when applied to deterministic data, produce uncertain results. For example, the LOCATE operator, which does pattern matching, may introduce uncertainty in the results, due to the exact nature of the data and the matching algorithms.

How to succinctly represent uncertain data and efficiently process it in databases has been an open problem for a long time. This is our main focus in dealing with value uncertainty. We have three ways of representing uncertain data values:

- **R1**: **Value-probability pairs**. An array value is represented as (v_1, p_1), (v_2, p_2),…, (v_n, p_n), where (v_i, p_i) indicates that the probability of the value being $\mathrm{v_i}$ is $\mathrm{p_i}$. If the sum of the p_i values, *psum*, is less than one, (*1-psum*) is the probability that the value does not exist in the array.
- **R2**: **An expectation and variance pair**. An array value, which can in general be the result of an operator, is represented as (E, Var), indicating the statistical information of the value. Compared to R1, although this is less informative, R2 is much more succinct and allows efficient processing of query operators. One can argue that the amount of uncertain information of R2 is sufficient for most applications of ASAP. For example, SUM or AVG on a huge number of values makes *individual* possible values unimportant; one would be concerned with the expected value and the variance.
- **R3**: **Upper and lower bounds**. Similar to R2, the statistical information is in the form of bounds: ([E], UB, $\mathrm{p_1}$, LB, $\mathrm{p_2}$), where E (expected value) is optional, and UB, LB are upper and lower bounds, respectively. It holds that $\Pr[\mathrm{v} > \mathrm{UB}] < \mathrm{p_1}$, and $\Pr[\mathrm{v} < \mathrm{LB}] < \mathrm{p_2}$. As with R2, the goal here is also efficient processing of query operators with an acceptable amount of uncertainty information returned to the end users.

R1 is similar to what has been proposed before in the literature [Bar92]; however, it does not scale well for most query operations. The query processing cost for generating a large number of value-probability pairs can be prohibitive. For example, SUM or AVG can cause the number of discrete values in a distribution to grow exponentially (all possible pairs), making it intractable. One alternative is to lower the granularity of the discrete probability points as query operators are applied. This, however, is still difficult if one wishes to obtain the same result value distribution.

ASAP's uncertainty model supports many options for how to represent values. The choice of representation has a very large impact on system performance. The main novelty of the ASAP approach is to trade accuracy in the uncertainty measure for ease of processing by allowing the system to choose among and convert between these various representations.

One way to alter representations is to choose among R1, R2, or R3. For example, a query on two arrays, each using R1, could more efficiently produce a result using R2 or R3. Such conversions between representations can be done as each query operator is applied. Not only these less-detailed representations seem to be sufficient and perhaps even more appropriate for most of our target applications, but also they make query processing on probabilistic data tractable (i.e., the cost is linear in the number of processed tuples).

Another strategy ASAP will use involves choosing the granularity at which to assign uncertainty values. For example, we can assign uncertainty to each cell in the array or to the array as a whole. In between these extremes, we can carve the array up into rectangles and assign an uncertainty measure to each such region. For example, we can divide a square array into 4 equal-sized quadrants, each with its own distribution. Each cell in a quadrant is thus assumed to have the same distribution "shape", only differing by their expected values.

6.2 Lineage Tracking

ASAP also contains a specialized lineage array that encodes the queries that have been entered to produce all target arrays, along with the source arrays they are derived from. In effect, this can be thought of as the derivation history of all materialized views. If done at the individual tuple level, tracking lineage can become very expensive and even NP-Hard for some operations [Wid05]. Thus, ASAP chooses to record only the processing (cooking recipe) that generated any given array. This capability also takes much less space and at present seems to be sufficient to meet the needs of many scientific applications that we are familiar with.

7. Now What?

The previous section indicated a collection of markets where a very noticeable performance differential can be realized with a custom architecture. It is obvious that the four markets have conflicting architectural requirements. Hence, they cannot be addressed with an OSFA DBMS.

As a result, there are several possible ways that DBMS architectures could evolve off in to the future:

- **Yawn (no change)**. One could argue that RDBMSs are fast enough to meet the needs of most of the customers in these four areas. Hence, there will be a few niche solutions to address the high end of any market, with RDBMS capturing the remainder of the customers.

  Although one might argue this point of view in markets 3 and 4, it is a bit difficult to make a serious case for this in markets 1 and 2. Especially in warehouse applications, where the data volumes and query complexity are going through the roof, this is a difficult position to defend.

- **K Systems united by a common parser.** One could argue that there will be some number, K, of engines, where K is determined by the number of non-trivial markets with specialized requirements. However, these can all be hidden under a common parser, with the actual user command directed to the correct engine.

  The effort to construct a common version of StreamSQL, which unites historical and streaming data, is a step in this direction. Whether this tactic can come to fruition in stream processing, let alone the other markets indicated in this paper, is anybody's guess.

- **K Systems using abstract data types**. Another possibility is to build what amounts to a complete engine within the extension systems present in current DBMSs. For example, a complete column store could be built as an extension. The net result is a different, less pleasant syntax, for something like the previous solution.

- **Data Federation**. One could simply come to grips with the fact that there will be a number of, basically incompatible, systems, and "adaptors" are required to map between them. This will be the "full employment act" for computer scientists for a long time to come, because mapping between different systems has proved semantically troublesome for the last 30 or so years, and shows no signs of getting easier anytime soon.

- **From Scratch Rewrite**. It is conceivable that a single code line could be architected with sufficient generality to encompass all of the requirements noted in this paper. For example, one could design a "morphing" ChunkyStore that could move between a row store and a column store, with various ChunkyStore alternatives in between. Such a storage system requires an optimizer and executor with dramatically more generality than current systems. It is conceivable that one could also construct a system that ran in client-server or embedded mode, and had some sort of "fast path" for text search.

The old adage comes to mind at this point "may you be blessed to live in interesting times". In our opinion, the next decade will be an interesting time to be active in the DBMS field as vendors cope with the choices laid out above. Also, there is an obvious charge to DBMS researchers; namely find an application area where OSFA does not work and figure out what does.

8. ACKNOWLEDGEMENTS

Parts of this work are supported by the NSF under grants IIS-0086057 and IIS-0325838.

9. REFERENCES

[Aba03] 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". In VLDB Journal, 2003.

[Aba06] D. Abadi, S. Madden, and M. Ferreira. "Integrating Compression and Execution in Column-Oriented Database Systems". In Proc. of the ACM

International Conference on Management of Data (SIGMOD), 2006.

[Apa06] http://www.progress.com/apama/index.ssp

[Ara04] 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 VLDB, 2004.

[Ast76] 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, June 1976.

[Bar92] D. Barbara, H. Garcia-Molina, and D. Porter. “The Management of Probabilistic Data”. IEEE Trans. Knowl. Data Eng., 4(5):487-502, 1992.

[Bon05] P. Boncz, M. Zukowski, N. Nes. “MonetDB/X100: Hyper-pipelining Query Execution”. In Proceedings of the Conference on Innovative Database Research (CIDR), 2005.

[Bre04] E. Brewer. “Combining Systems and Databases: A Search Engine Retrospective,” in Readings in Database Systems, M. Stonebraker and J. Hellerstein, Eds., 4 ed, 2004.

[Cha06] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. “Bigtable: A Distributed Storage System for Structured Data”. In Proc. of the Conference on Operating System Design and Implementation (OSDI), 2006.

[Cor06] http://www.coral8.com/

[Dea04] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In Proceedings of the Conference on Operating Systems Design and Implementation (OSDI), 2004.

[Ghe03] S. Ghemawat, H. Gobioff, and S.-T. Leung. “The Google File System”. In Proceedings of the nineteenth ACM SOSP, 2003.

[Gra06] C. Graham. Market Share: Relational Database Management Systems by Operating System, Worldwide, 2005”. Gartner Report No: G00141017, May 2006.

[Har06] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. "Performance Tradeoffs in Read-Optimized Databases." In Proceedings of the 32nd Very Large Databases Conference (VLDB), 2006.

[Hdf06] http://hdf.ncsa.uiuc.edu/HDF5/

[Mat06] http://www.mathworks.com/

[Mot03] 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 Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003), Asilomar, CA, 2003.

[Net06] http://www.unidata.ucar.edu/software/netcdf/

[Ola06] OLAP Market Report. Online manuscript. http://www.olapreport.com/market.htm

[One06] P. O’Neil, E. O’Neil, and X. Chen. “A Star Schema Data Warehouse Benchmark”. Online Manuscript. http://www.cs.umb.edu/~poneil/publist.html

[Pik05] R. Pike, S. Dorward, R. Griesemer, S. Quinlan. “Interpreting the Data: Parallel Analysis with Sawzall”. In Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure 13:4, pp. 227-298.

[Sar94] S. Sarawagi and M. Stonebraker. “Efficient Organization of Large Multidimensional Arrays”. In Proceedings of the 10th International Conference on Data Engineering (ICDE), 1994.

[Seq93] http://s2k-ftp.cs.berkeley.edu:8000/index.html

[Sto05a] M. Stonebraker and U. Cetintemel. “One Size Fits All: An Idea Whose Time has Come and Gone”. In Proceedings of the International Conference on Data Engineering (ICDE), 2005.

[Sto05b] M. Stonebraker, D. J. 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, S. Zdonik. “C-Store: A Column Oriented DBMS”. In Proceedings of the Conference on Very Large Databases (VLDB), 2005.

[Sto76] M. Stonebraker, E. Wong, P. Kreps, and G. Held. “The Design and Implementation of Ingres”. ACM Journal on Transactions on Database Systems (TODS), 1(3), 1976.

[Stre06] http://www.streambase.com/

[Ver06] Vertica Inc. http://www.vertica.com/

[Wid05] J. Widom. “Trio: A System for Integrated Management of Data, Accuracy, and Lineage”. In Proc. of the International Conference on Innovative Database Research (CIDR), 2005.