DeltaDB: A Scalable Database Design for Time-Varying Schema-Free Data

Peter Ivie and Douglas Thain
Department of Computer Science and Engineering, University of Notre Dame
{pivie,dthain}@nd.edu

***Abstract*—DeltaDB is a model for a database consisting of records with no fixed schema whose entire history is captured over time. It is designed to support efficient queries against the current state of the database, any point in the history of the database, and historical data aggregations over time. In this paper, we present the DeltaDB data model, the associated query algebra, and highlight the fundamental query optimization concerns. To gain experience with the DeltaDB model, we have created a single-node implementation of the database and used it to collect one year's worth of monitoring data from a distributed software system, reducing over 5TB of snapshots into 11GB of log data. We give examples of novel types of queries that exploit the time-varying nature of the data and evaluate their performance. We conclude with a discussion of how the single-node implementation will serve as the building block for a future distributed implementation.**

I. Introduction

DeltaDB is a model for a database consisting of records with no fixed schema whose entire history is captured over time. It is designed to support efficient queries against the current state of the database, any point in the history of the database, and historical data aggregations over time. DeltaDB is suitable for datasets such as scientific data repositories, computer system monitoring, and sensor networks [1], where the data schema is likely to evolve, and the time-variant nature of the data is a primary concern.

Historically, relational databases have been implemented using the current state as the primary storage format, relying on fixed field widths to allow for updating of values in place. In such a scheme, a small log is often added to improve update performance or to implement transactions. The efficiency of queries is improved by adding indexes in the form of additional tables that must be updated along with the primary data.

DeltaDB takes a different approach. Because the time-varying nature of the data is the primary concern, the log of updates is the *primary data structure* in the database. Appending to a log does not require fixed field widths, which enables schema-free data with no declaration of fixed field widths in advance. In the manner of other big-data systems, acceleration of queries is accomplished by the liberal application of parallel hardware, rather than by the construction of indexes against the primary data structure.

Our design for DeltaDB begins with a description of the data model and the logical log. We describe a time-variant algebra that is similar to the relational algebra but adds several operators necessary for dealing with time-variant data and aggregating across both time and space. We follow with examples of how common queries in the form of a calculus are mapped to the algebra.

To gain experience with the DeltaDB data model, we have created a single-node implementation (LibDeltaDB) and used it in production to record the vital statistics of servers in a distributed system over the course of one year, recording over 193 million updates and reducing over 5.2 TB of time variant data down to 11.2GB of logs. While the amount of data collected by this prototype is modest compared to other big-data systems, it has allowed us to experiment with the DeltaDB query model and understand the performance implications of various query forms.

We conclude with a sketch of how DeltaDB could be used to implement a much larger distributed database system where the single-node LibDeltaDB serves as the building block for each node in the system. By partitioning and replicating the records across multiple instances of LibDeltaDB, the same update and query semantics can be achieved through the proper application of the DeltaDB algebra.

As an aside, much recent work on big data systems has abandoned the traditional formalities that were developed and applied to the relational database model [2]. (See Stonebraker [3] for a comprehensive criticism.) We believe that these formal tools – with modifications – remain effective tools for designing, describing and implementing big data systems. As we show below, the relational algebra is an effective tool for expressing how complex queries can be decomposed across a distributed system while enabling a rational discussion of correctness and efficiency.

II. Data Model

Figure 1 shows the logical structure of a DeltaDB database. A database consists of an unordered set of **records** each with a unique **key** selected by the system. Each record contains a set of **attribute-value pairs** with no fixed schema. JSON is used for the external representation of records. The basic update operations on the database are given in Figure 2.

DeltaDB is a schema-free database, by which we mean that there are no enforced constraints upon the fields in each record. Over time, the attributes of a record may be added or removed, change value, or even change type from an atomic value, to a list, then to a set. Obviously, the user of the database must establish some degree of consistency in order to retain the value of a dataset, but the database allows the schema to evolve as needed.

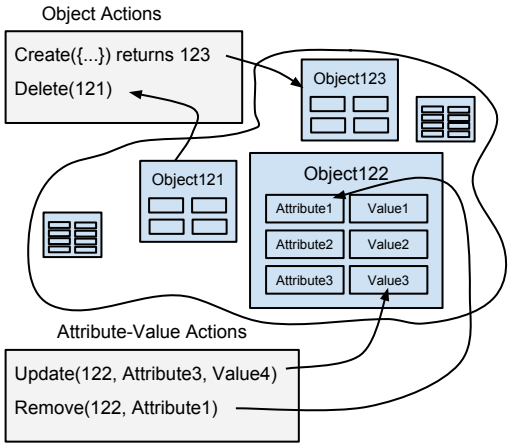


Fig. 1. DeltaDB Data Model

```
// Create a new record, yielding the unique key.
Create( record ) returns key;

// Delete an entire record, given the unique key.
Delete( key );

// Update one attribute in an existing record.
Update( key, attribute, value );

// Remove an attribute from an existing record.
Remove( key, attribute );
```

Fig. 2. DeltaDB Update API

The database keeps the entire history of updates to every record. When a record (or attribute) is removed, the removal is noted, but the historical records remain. A **snapshot** is the output of records at a specific point in time. A query against the database can retrieve a snapshot, or a **history** that yields the evolving state of selected records over time.

Conceptually, it is useful to think of a DeltaDB database as consisting of nothing but a time-sorted log of changes or **deltas**. Deltas that could be included in the log file include the creation and deletion of objects, and the removal and update of attribute-values. Also included in the log are **timestamps** which indicate the wall clock time of following deltas until the next timestamp. The order of events in the log is what is causally significant – events with the same timestamp occur when updates happened faster than the system's clock update interval. Figure 3 shows an example of a log consisting of four deltas, and how the database and the corresponding snapshots that could be viewed after each delta was applied.

By itself, a bare delta log is not an efficient query structure. The state of any given record could potentially be spread across the entire timespan of the log, if it was created near the beginning, updated in the middle, then deleted at the end. To reconstruct a snapshot of a record at any point in time, a query would have to process the entire log from the beginning until the desired timestamp was reached.

A realistic implementation of DeltaDB must make use of a series of **checkpoints** which are simply stored snapshots. Between each checkpoint is a delta log reflecting the sequence of updates until the next checkpoint. This permits a query to start at the checkpoint preceding the desired query interval, and then end as soon as the ending timestamp is reached. Changing the frequency of checkpoints allows one to exchange storage space for query performance. In Figure 3, a checkpoint frequency of 6 would only store Snapshots #1 and #3. If Snapshot #2 were to be needed for a query, it would need to be generated from Snapshot #1 and Delta #2.

III. Query Algebra

To perform queries against DeltaDB, we define a query algebra consisting of six operators, summarized in Figure 4. While similar to Codd's relational query algebra, the semantics differ significantly to accommodate the time-varying nature of the data. Each operator accepts a common data format as input: a checkpoint giving the data state at the starting time of the query, followed by a stream of deltas until the ending time. The operators may be applied in any order to achieve the desired query semantics or performance, except for τ and ν which must be first and last, respectively.

Some operators must keep internal state, so that when a delta arrives on the input, the rest of the corresponding record is available to act upon. The amount of state maintained is essential to understanding the inherent efficiency of a given query. Reordering the operators may result in significant gains in efficiency by reducing both state and communication between operators.

Temporal Collector: $\tau(t_1, t_2)$ The temporal collector accesses the raw data on disk and emits a checkpoint of the database state at t_1 followed by a stream of all deltas until t_2. As noted above, the raw data may contain periodic checkpoints, allowing τ to begin reading from a timestamp preceding but close to t_1.

Selection Operator: σ(expr) The selection operator evaluates a given expression (name="John") on every record in the input stream. If the expression matches the record, it is passed as output, otherwise it is discarded.

Two cases must be considered based on the attributes in the expression. A *static* attribute is defined once at record creation and never changes, while a *dynamic* attribute changes throughout the lifetime of a record. If all attributes of interest in the expression are *static*, then σ is stateless and simply evaluates one record at a time independently. If any attribute is *dynamic*, then σ must keep state equal to the input checkpoint, apply deltas to its internal state, and re-evaluate the expression after applying each delta.

Projection Operator: π(expr-list) For each checkpoint or delta on the input stream, the projection operator evaluates its expressions, deriving a respective checkpoint or delta for the output stream. π must maintain the current state of each record in the stream. Closely connected with the π are the reducers ϕ and ψ, which summarize attribute values across time and records respectively.

Temporal Reducer: $\phi(\text{attr}, \text{func}, \text{time})$ The temporal reducer summarizes the data from *each record individually* over a given time-span. The first argument gives the attribute, the second the reducing function, and the third the granularity for partitioning time. The output of ϕ is a checkpoint followed by a series of deltas at regular time intervals. If the time-span is equal to the duration of the log, then all values would be reduced to a single record on the output. The time granularity

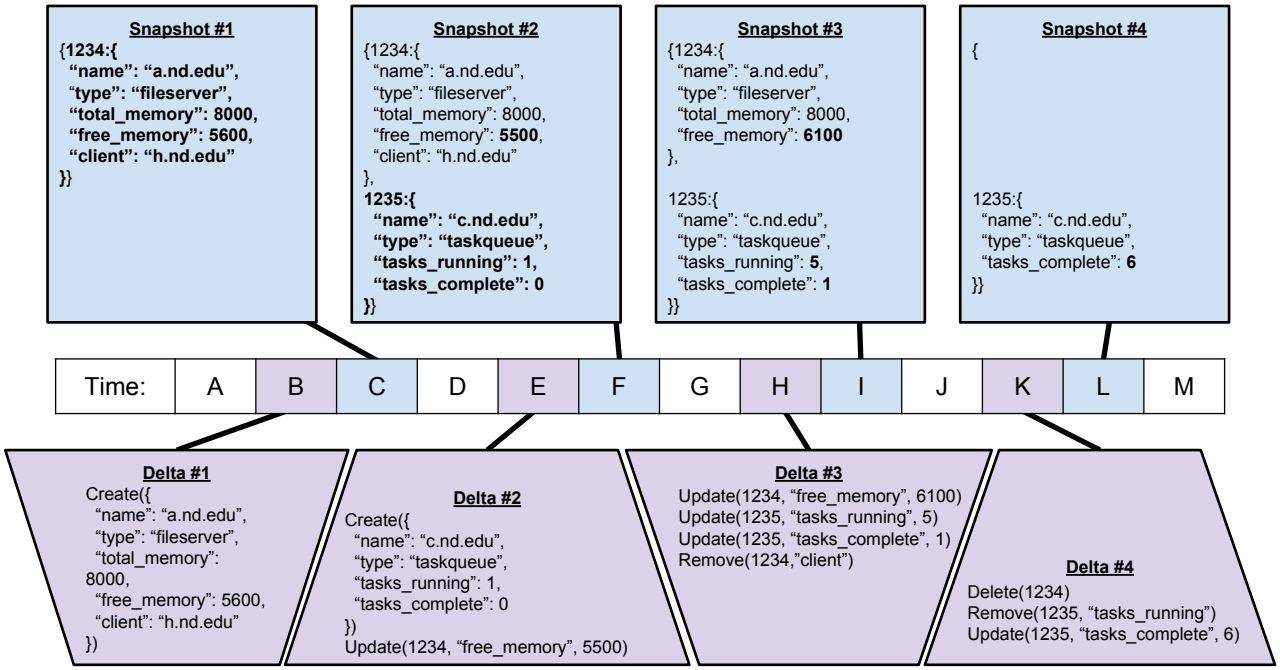


Fig. 3. Comprehensive Checkpoints and Deltas

Name	Symbol	Description	State required
Temporal Collector	$\tau(t_1, t_2)$	Select records in a time range.	One checkpoint until start time, then one delta
Selection Operator	σ(expr)	Select records matching an expression.	One checkpoint plus one delta
Projection Operator	π(record-expr)	Compute new values from input records.	One checkpoint plus one delta
Temporal Reducer	ϕ(attr, func, time)	Reduce values within time spans	One checkpoint plus all deltas during Time Span
Spatial Reducer	ψ(attr, func)	Reduce values from multiple objects	One checkpoint plus one delta
Pivot Operator	ν(attr-list)	Convert to tabular format.	One checkpoint plus one delta

Fig. 4. Time-variant Algebra Operators

is entirely independent of the frequency of deltas on the input stream. If the output frequency is much smaller than the input frequency, more output records will be generated than when the output frequency is much larger.

The reducing function must be chosen with care, since the number of deltas over a given time period could be any arbitrary number (even zero) and are not necessarily evenly distributed over time. Thus, COUNT and SUM are not likely to be good temporal reduction functions, because they are sensitive to the input delta frequency. However, functions such as as MAX, MIN, AVERAGE, FIRST, LAST etc. are not sensitive to update frequency.

As an example, ϕ(temp, PAVG, 60s) considers deltas for the first 60 seconds. The values for $temp$ are averaged proportionally based on the percentage of seconds during which each value was valid. That proportional average is streamed out as a single value for $temp$. Then the operator considers each following 60 second spans in order in the same manner.

Spatial Reducer: ψ(attr, func) The spatial reducer summarizes *all records at a given time* for each time point in the input stream. The output is a single summarized record for each delta on the input stream. The function may be any common reduction function such as MAX, MIN, SUM, AVERAGE, COUNT, etc.. Because the set of objects in the database is (spatially) unordered, the reduction function must not be order-sensitive such as FIRST or LAST.

As an example, ψ(temp, AVG) would average the temperatures across all records in the database, yielding a stream of average temperatures, computed each time any member of the average was changed.

Pivot Operator: ν(attr-list) The pivot operator converts the record data into a tabular form that is more readable and is suitable for automated plotting. ν appears as the last operator in the chain, or not at all. For example, ν(time,temp), would produce a table with two columns – time and temperature – with a row in the table for each delta on the input.

IV. Query Optimization

Two main approaches can be taken in an attempt to improve the basic operators defined above. One significant consideration is the extent to which the types of arguments in the selection operator can affect efficiency. Another consideration is the order that operations are applied to minimize data transfer and memory requirements. These two considerations will be discussed below. Other optimizations are likely to exist, but will not be addressed here.

A. Selection operator

Anticipation of attribute behavior within objects can prevent streaming delays between operators.

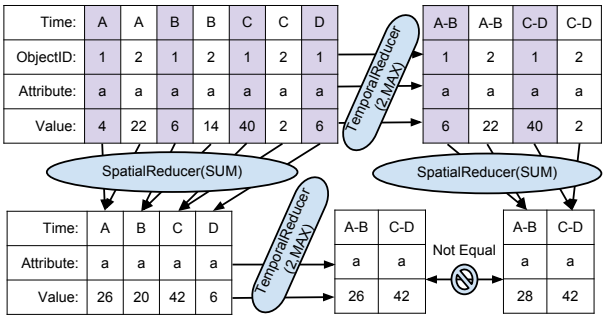


Fig. 5. Example of projection and reduction

Static attributes: These attributes are assigned a value when the object is created and are never changed by a delta. This property makes the selection operator more efficient because evaluation of the expression (defining which objects to filter) is complete when the object is first introduced. Since the value will never change, evaluation of an expression involving only static attributes will never change. The operator only needs to keep track of which objects to filter.

Dynamic attributes: If the value for an attribute used in the expression changes, the selection operator must re-evaluate the expression for each delta. To do this, the operator must maintain the state for each object. If the result of the expression changed after applying the delta, the object needs to be created or deleted in the output as appropriate.

Super-dynamic: A more expensive approach to dynamic attributes could allow the user to eliminate an entire object if the expression is false even once during the lifetime of the object. This approach leaves the operator with two options: 1) Make two passes through the data, one to identify which objects to include, and another to apply the selection, and 2) Load all state into memory, and only start output until all expressions have been evaluated at every delta. If this type of query is absolutely necessary, some work might be done to preprocess the data in order to make it possible to identify, in advance, a list of object keys to include, thus preventing streaming delays.

B. Order of Operations

The operators are designed to place no logical restriction on the order in which they are applied. However, the order in which they are applied can have a significant impact on performance and scalability. Figure 4 enumerates the amount of state each operator needs to store in memory.

A full projection based on only the attributes needed in the final result could eliminate attributes needed to evaluate later expressions. But in some cases a partial projection could occur before a selection and the remaining fields could be removed later on with a full projection. The projection operator is not supported in most commercial database implementations either [4], since the operator can be applied as needed as when the desired output fields are specified.

The temporal reducer has a worst case when the time-span is greater than or equal to the entire duration of the log. In this case all state must be stored in memory. In the best case, with a time-span of zero, this operator needs only a single snapshot and a single delta in memory at a time.

The spatial reducer, on the other hand, is able to stream results with every delta, so it always needs only a single snapshot and single delta in memory at a time.

However, consideration of the example data in Figure 5 should also make it clear that reversing the order of the temporal and spatial reducers provides a different result. Therefore the user may need control over which of these two operators is performed first.

Figure 5 also illustrates the process of applying a pivoting operator, temporal reduction, and spatial reduction. The pivoting operator would normally be applied as the last operator in a query, but it is shown as if it was applied earlier, in this case, to make the behavior of the reducers more clear.

V. Single Node Case Study

We have created a single-node prototype of DeltaDB called LibDeltaDB. The prototype consists of a C library which implements the create, delete, update, and remove functions by logging to local disk. A checkpoint file enumerates the state of all records at the beginning of each day. The log data for each day is stored in a separate file. We have implemented each query operator as a separate Unix process. The various operators are piped together in order to implement complete queries. This strategy offers acceptable performance for a single-node implementation, and facilitates the distributed query structures described in the following section.

To gain experience with LibDeltaDB, we have used it in production for over a year to provide system monitoring services for the Cooperative Computing Tools, a suite of software developed at the University of Notre Dame. Briefly, every service running in the distributed system sends periodic messages to a central catalog server, recording the server name, location, operating system resources, software version, clients connected, and so forth. The services involved include the Chirp [5] distributed filesystem and the Work Queue [6] master-worker framework.

The catalog server uses LibDeltaDB to record updates from services and provide real-time queries to system participants. This enables tracking of where the software is installed, what performance is achieved, and what versions are in use. The current snapshot is cached in memory to provide rapid response to the most common queries on the current state, while the historical data is available to system administrators. DeltaDB is well suited for this task, because it accomodates the changes in schema that come with evolving software, while at the same time facilitating historical analysis of the deployed services for project management.

Figure 6 highlights some vital statistics of this instance of DeltaDB. The service has been running for over one year, generating 11.2GB in log files. A complete query can process about 3.26 days of log data per second. On average, each record in the database is updated every 255 seconds. There are only 3 globally static attributes that are never updated in any record: name, key, and address.

Statistic	Value
Average log data per day (bytes)	29,138,676
Average updates per day	530,139
Timestamps with deltas	76%
Average snapshot size (bytes)	216,273
Total number of attributes	89
Number of attributes in an object	7-56
Average number of attributes per object	24

Fig. 6. Statistics for Single Node LibDeltaDB

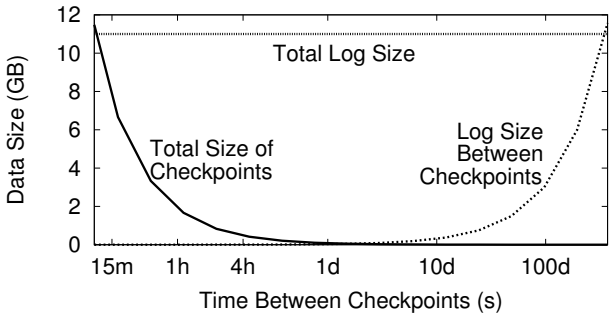


Fig. 7. Snapshot storage tradeoff

A. Storage Tradeoffs

For queries involving only a portion of the data, periodic checkpoints can have a significant positive impact on performance. For any query, the ideal checkpoint would be one just before the starting time of interest. Having a snapshot for every second when the database state changes, however, would require approximately 5.2TB per year.

But if checkpoints are less frequent, there is more log data in between them. A single point query will have to parse half of that log data on average. However, checkpoint frequency can easily be adjusted to balance performance and disk use. Figure 7 shows that with 1 snapshot per day, both the space consumed by snapshot data, and the log data between snapshots is small compared to the total log size.

If many queries involve a single point or small period of time, more frequent snapshots could help query performance. But at about 10 minutes between time stamps, the snapshots would take up as much space as the log data itself.

B. Example Queries

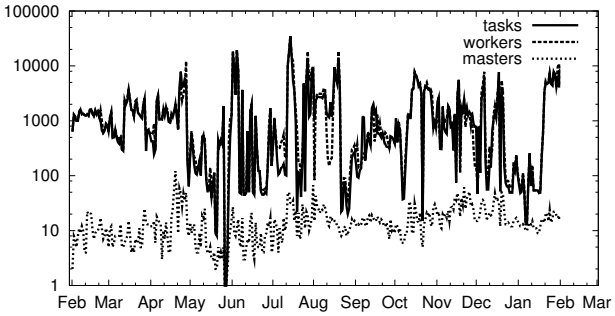
Figure 8 gives some examples of time-based queries and the equivalent query algebra. (Note the notional examples of how these might be expressed in SQL, but a query language has not been implemented on top of the query algebra.)

Query #1 summarizes all task queues at 15 minute intervals over 1 year. 1 year of data is read from disk, starting on 1 Feb 2013. Objects with attribute 'type' not equal to 'queue' are filtered out. Then for each 15 minute span after the start time, the highest value occurring during that span is computed for the attributes 'workers' and 'tasks'. Then, in each span, the number of objects is counted (with 'name,CNT'), and the maximum values for 'workers' and 'tasks' are added together. For each 15 minute time span, a row is streamed back to the user with the three values described previously.

Example Query #1:
List the total number of queues, workers, and tasks at 15 minute intervals over the course of one year.

```
SELECT COUNT(name), sSUM(tMAX(workers)),
       sSUM(tMAX(tasks)) WHERE type=queue
AT '2013-02-01 00:00:00' PLUS 365 DAYS
BY SPANS OF 15 MINUTES
```

τ 2013-02-01@00:00:00 d365 |
σ type="queue" |
ϕ m15 workers,MAX tasks,MAX |
ψ name,CNT workers.MAX,SUM tasks.MAX,SUM |
ν name.CNT workers.MAX.SUM tasks.MAX.SUM



Example Query #2:
For each month of the year, list each project run in that month, with the owner and number of tasks dispatched.

```
SELECT project, owner, tMAX(workers),
    tMAX(dispatched), tMAX(tasks)
WHERE type="queue"
AT '2013-02-01 00:00:00' PLUS 365 DAYS
BY SPANS OF 1 DAY
```

τ 2013-02-01@00:00:00 d365 |
σ type="queue" |
ϕ d30 workers,MAX dispatched,MAX tasks,MAX |
ν project owner workers.MAX dispatched.MAX tasks.MAX

Example Query #3:
List the names of the file servers that were in operation on February 3rd, 2013.

```
SELECT tLAST(name) WHERE type=fileserver
AT '2013-03-13 00:00:00' PLUS 1 DAYS
BY SPANS OF 1 DAY
```

τ 2013-03-14@00:00:00 d1 | σ type=fileserver |
π name | ϕ d1 name,LAST | ν name.LAST

Fig. 8. Example Queries

Query #2 summarizes individual work queue status at 30 day intervals over 1 year. This query starts out the same as the first one, but the time span is 30 days instead of 15 minutes. Another difference is that without a spatial reducer the pivot operator returns a row for each object before it moves on to report the rows for the next time span.

	Query #1		Query #2		Query #3	
	Output	Time	Output	Time	Output	Time
cat	11.2GB	1:13	11.2GB	1:13	63.8MB	0:0.30
τ	11.2GB	1:44	11.2GB	1:44	31.9MB	0:0.28
σ	313MB	1:49	312MB	1:48	31.1MB	0:0.32
π	-	-	-	-	34.7KB	0:0.35
ϕ	40MB	1:49	56KB	1:48	29.4KB	0:0.34
ψ	3.6MB	1:50	-	-	-	-
ν	1.5MB	1:51	12KB	1:52	9.3KB	0:0.35

Fig. 9. Performance of Example Queries

Query #3 simply lists file servers that were active on a certain day. This query is no less complex than Query #2, but faster because only one day is considered. Since the time span is also one day, the result is a single row for each object over a single time span.

C. Performance

Each of the three sample queries were executed on an i5-3570 CPU at 3.4Ghz with 8GB of RAM and the data was stored on a Seagate Barracuda ST500DM002. The size of the checkpoints and logs (at over 11.2GB) is larger than the size of RAM, so the data cannot be fully cached in memory. While in this case we could procure more memory, we want to evaluate the out of core case.

The 'cat' command was executed on the relevant log files as a reference for the performance of the hard drive. Then only the first operator was performed for each query while space and time requirements were measured and recorded. The results were sent to /dev/null for the time measurement, and piped to 'wc' for the space measurement. Then the following operators in each query were added one by one and performance was again measured and recorded.

Figure 9 indicates that the bulk of each query is just getting the data from disk. It is not unusual for Big Data approaches to be mostly disk bound. The τ operator only performs 44% slower than a blockwise read (using 'cat') with no processing. The successive operators require a minimal amount of additional time to perform their operations.

And so we conclude that this approach is mostly limited by disk speed. Keeping each operator streaming to the next one makes it so that by the time the last of the log data is read from disk, most of the data before it has already been processed to the end of the chain of piped operators. In this manner, the query can be performed almost as quickly as the necessary data can be read from disk.

VI. Distributed DeltaDB

DeltaDB can be implementation as a distributed system in order to increase storage capacity, query load, and/or availability beyond a single node's capacity. We have not yet built a distributed implementation, but in this section we present the key design issues related to a distributed DeltaDB.

In the most straightforward implementation, a distributed DeltaDB would consist of a head node and multiple storage nodes. Each of the storage nodes would be implemented using LibDeltaDB as the fundamental storage engine. Updates would pass through the head node to the proper storage node, while

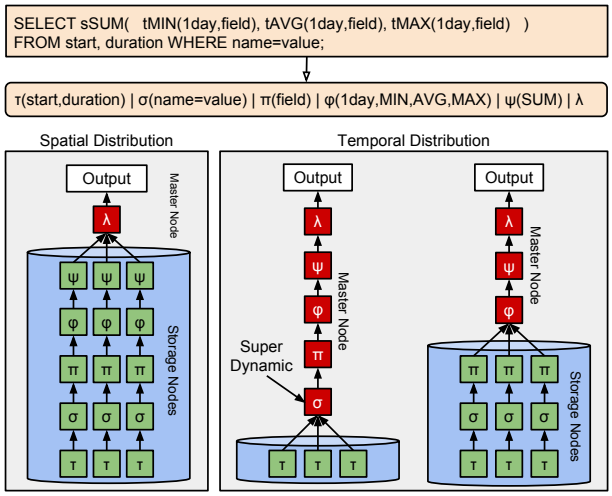


Fig. 10. Operator distribution examples

	Distributable	
Operator	Temporally	Spatially
Selection (σ)	For Static	Always
Projection (π)	Always	Always
Temporal Reduction (ϕ)	Partially	Always
Spatial Reduction (ψ)	Always	Mostly

Fig. 11. Operator distributability

queries would be implemented by distributing the various operators across the system, connected by network pipes.

The key design decision of how to distribute data across the storage nodes naturally affects both update and query performance. Temporal and spatial approaches may be considered. In both cases, replication of data is necessary and desirable, but does not affect the fundamental strategies.

As shown in Figure 10, queries against a distributed database are implemented by distributing the query operators across the storage nodes and the master node. The former may be run in parallel, while the latter are serialized, so it is clearly desirable to select a data layout that maximizes the portion of the queries that are performed by the storage node. An operator is considered distributable (see Figure 11) when virtually all of the effort can be performed on storage nodes.

A. Temporal Distribution

Under temporal distribution, all data from a given time range is stored together on a single node. (e.g. The checkpoint and log for day 1 are stored on node 1, day 2 on node 2, etc.) The mapping from time range to storage node could be accomplished through consistent hashing techniques, or simply storing a map at the master node. As the database increases in size, capacity is easily increased by adding nodes without requiring data redistribution.

Database updates are easily handled: the master simply sends all updates to the currently active storage node, where they are logged. However, this has the downside that update performance under temporal distribution is no better than the single node implementation.

On the query side, temporal distribution is most effective for supporting queries that scan the entire time range supported

by the database. In that case, a temporal selector can run on every storage node of the system in parallel. If a query only addresses a short time range, only the storage node(s) representing that time will be addressed.

Selection Operator (σ): A selection operator is fully distributable to individual nodes as long as the super dynamic situation described in section 4.1 can be avoided. In that case, storage nodes might be able to reduce network traffic by applying the selection operator, but the head node would have to perform the operation again to see if additional objects need to be filtered. This underscores the importance of avoiding this situation if possible.

Projection Operator (π): Projection is fully distributable because it only needs to consider immediate attribute names. It does not require knowledge outside of each creation of an object or the update/removal of an attribute.

Temporal Operator (ϕ): If the time span boundaries in the query happen to line up with the time period boundaries on the storage nodes, then the temporal operator is distributable. Otherwise inner time spans could be processed by the storage nodes, but the partial time spans split across nodes must be combined by the head node and processed there. In the worst case, where the time span is infinite, no work can be performed by the storage nodes.

Spatial Operator (ψ): This operator can be fully completed on the storage node as long as all the previous operators were performed on the storage node also.

B. Spatial Distribution

Alternatively, data could be spatially distributed across nodes within the constraint that all updates to a given record must be logged at the same node, so that operators applied to that log have the entire history. The simplest spatial distribution strategy would be to hash on the unique key. Alternatively, the master could hash on a *static* property of a record, such as name or address in order to partition the data by content. As with temporal distribution, consistent hashing techniques applied in the master would allow for reconfiguration of the system without data redistribution.

With spatial distribution, update traffic is distributed across the system, offering the potential for increased update throughput by multiple clients.

Under this distribution, the temporal range of queries no longer has an effect on which storage nodes are needed for a query. If the record key is used to distribute records, then all nodes must participate in all queries. If the data is partitioned by content, then the presence of a selection operator on a static attribute will limit the set of nodes accordingly.

Selection Operator (σ): Because all the data for each object is stored on a single node, the selection operator has access to all the data it needs on the storage nodes.

Projection Operator (π): Projection is again fully distributable because it only needs to consider immediate attribute names. It does not require knowledge outside of each creation of an object or the update/removal of an attribute.

Temporal Operator (ϕ): Temporal reduction is fully distributable. Even for an infinite time span, all data for a given object is available on a single storage node.

Spatial Operator (ψ): This operator is the most complex. Take, for example, a situation where one node has only one object matching the selection criteria, and another node has many objects. Some of the spatial reduction operators are clearly fully distributable, such as MIN, MAX, SUM, and COUNT. For each of these operators, once the work has been done on the storage nodes, the remaining work of combining the results is trivial.

Performing an averaging reduction is a little more complex, but can still be performed with minimal work after the initial results. Sum and count operators can be pushed to the individual nodes, aggregated on a master node, and then the final result can be obtained simply by dividing the sum by the count. A weighted average or some other specialized reducers might not be as distributable however.

C. Implementation on MapReduce

The DeltaDB operators could be implemented in the MapReduce framework to operate on these files. The same distributability of operators would apply as shown in Figure 11. Using a single reducer in the final output would maintain the time sensitive order of the results.

A few adjustments would need to be made to the file structure of LibDeltaDB. Hadoop divides files into blocks for distributed storage and processing. LibDeltaDB could keep files below 64MB and put a checkpoint at the beginning of each file to preserve consistency.

For the case study with temporal distribution, up to 2.189 days worth of log data would fit into each file with the checkpoint data. This would make each file self contained and capable of being processed completely in parallel.

With spatial distribution, the checkpoint would be smaller because the state of fewer objects must be declared. For the case study, the checkpoint is already insignificant in size compared to the log data, but in a situation where there are more objects and fewer updates, this difference would become more pronounced. This points to spatial distribution as having more potential for scalability.

VII. Related Work

The concepts used to design DeltaDB and implement LibDeltaDB are not new. The database attributes described in this section are critical for DeltaDB. When viewed individually, they are mostly available in other databases (see Figure 12). But existing databases have not been designed to handle all of them together.

Log-Only: Write-ahead logging (WAL) [7] is a common approach in databases, i.e. MySQL, DB2, PostgreSQL [8], MongoDB [9], BigTable [10], HBase, Cassandra [11], and is similar to the journaling feature in file systems. The idea is to get all data modifications saved to disk as quickly as possible so that data loss is minimal in the event of a system failure. As shown in the case study above, it can also be a very efficient way to store data in terms of space consumed.

LogBase [12] retains a relational data model, but like DeltaDB, it keeps the data permanently in this log and performs queries directly on this data. Most other databases

	Log-Only	Schema-Free	Multi-Version	Temporal-Reduction
Oracle	No	No	No	No
DB2	No	No	Yes	No
MySQL	No	Add	No	No
PostgreSQL	No	No	Add	No
MongoDB	No	Yes	Yes	No
Cassandra	No	Yes	Yes	No
HBase/BigTable	No	Yes	Yes	No
Oracle NoSQL	No	Yes	No	No
Oracle Workspace Manager	No	No	Yes	No
LogBase	Yes	No	Yes	No
TempoDB	?	No	Yes	Yes
DeltaDB	Yes	Yes	Yes	Yes

Fig. 12. Databases with desired features

incur extra overhead by searching for another location for the modification and storing it again there. The other location is chosen in order to optimize certain performance characteristics that each database is geared towards. For systems where queries are a bottleneck, this can be appropriate. But for systems where write-throughput is a bottleneck, this reordering of data imposes a limit on throughput.

Schema-Free: The entity-attribute-value (EAV) model [13] can be used to handle schema-free data in any relational database. But it often leads to queries with inefficient multi-attribute expressions. NoSQL databases, however, are designed to handle schema-free data natively.

Oracle NoSQL is designed specifically for schema-free data, but it is separate from their well-known database solution. Custom storage engines for MySQL enable it to accept schema free data [14] [15], but no official implementation exists.

Multi-Version: The main feature in ANSI/ISO SQL:2011 is support for temporal databases. IBM DB2 [16] has implemented this functionality allowing multiple historic versions of a record in addition to the current version. It also handles proposed versions of a record that automatically take effect at a specified time in the future. Oracle Workspace Manager [17] has similar capabilities, and a package built for PostgreSQL [18] enables temporal capabilities [19].

Any schema-free database could handle multi-version data by building the version into the attribute names, but the attribute names could quickly get confusing.

Temporal-Reduction: In an overview [20] of time-series data solutions Chukwas, OpenTSDB, TempoDB and Squwk are compared and TempoDB is recommended. TempoDB provides range rollups, which are similar to temporal reduction in LibDeltaDB. However, it does not support attributes grouped together into objects, and so cannot support spatial reduction.

Potential Features: In an open distributed system having a history of changes can be helpful in identifying tampering [21]. Also, stream data bears some similarity to DeltaDB and research in that area [22] [23] might be applied to optimize LibDeltaDB queries. The "store locally, query anywhere" paradigm used in GaianDB [24] would allow for elasticity in a distributed DeltaDB.

VIII. Acknowledgments

This work was supported in part by the National Science Foundation under grants PHY-1247316 and OCI-1148330, and the Department of Education under grant P200A120206.

References

[1] L. Ramaswamy, V. Lawson, and S. V. Gogineni, "Towards a quality-centric big data architecture for federated sensor services," in *Big Data (BigData Congress), 2013 IEEE International Congress on*, pp. 86–93, IEEE, 2013.

[2] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, pp. 377–387, June 1970.

[3] M. Stonebraker, "SQL databases v. NoSQL databases," *Communications of the ACM*, vol. 53, no. 4, pp. 10–11, 2010.

[4] D.-K. Burleson, *Inside the database object model*. CRC Press, 1999.

[5] D. Thain, C. Moretti, and J. Hemmes, "Chirp: A Practical Global Filesystem for Cluster and Grid Computing," *Journal of Grid Computing*, vol. 7, no. 1, pp. 51–72, 2009.

[6] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain, "Work queue+ python: A framework for scalable scientific ensemble applications," in *Workshop on Python for High Performance and Scientific Computing at SC11*, 2011.

[7] J. Gray, "A transaction model," in *Automata, Languages and Programming*, pp. 282–298, Springer, 1980.

[8] "PostgreSQL (Write-Ahead Logging)." http://www.postgresql.org/docs/9.1/static/wal-intro.html, 2014. [Online; accessed 24-February-2014].

[9] K. Chodorow, *Scaling MongoDB*. O'Reilly, 2011.

[10] 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," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[11] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[12] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "LogBase: a scalable log-structured database system in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1004–1015, 2012.

[13] P. M. Nadkarni, "Data extraction and ad hoc query of an entity-attribute-value database," *Journal of the American Medical Informatics Association*, vol. 5, no. 6, pp. 511–527, 1998.

[14] B. Taylor, "How friendfeed uses mysql to store schema-less data." http://backchannel.org/blog/friendfeed-schemaless-mysql, 2009. [Online; accessed 21-February-2014].

[15] "Schema-Free MySQL vs NoSQL." http://www.igvita.com/2010/03/01/schema-free-mysql-vs-nosql/, 2014. [Online; accessed 27-February-2014].

[16] IBM, "DB2." http://www.ibm.com/developerworks/data/library/techarticle/dm-1204whatsnewdb210/index.html#ttq, 2014. [Online; accessed 14-February-2014].

[17] Oracle, "Workspace manager." http://www.oracle.com/technetwork/database/enterprise-edition/index-087067.html, 2014. [Online; accessed 14-February-2014].

[18] S. Deckelmann, "Temporal PostgeSQL." http://pgfoundry.org/projects/temporal/, 2014. [Online; accessed 14-February-2014].

[19] R. T. Snodgrass, "Temporal databases," in *IEEE computer*, Citeseer, 1986.

[20] T. W. Wlodarczyk, "Overview of time series storage and processing in a cloud environment," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pp. 625–628, IEEE, 2012.

[21] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr, "Distributed time-aware provenance," in *Proceedings of the 39th international conference on Very Large Data Bases*, pp. 49–60, VLDB Endowment, 2012.

[22] L. Cao and E. A. Rundensteiner, "High performance stream query processing with correlation-aware partitioning," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, 2013.

[23] A. Haque, B. Parker, and L. Khan, "Labeling instances in evolving data streams with mapreduce," in *Big Data (BigData Congress), 2013 IEEE International Congress on*, pp. 387–394, IEEE, 2013.

[24] "GAIAN Database." https://www.ibm.com/developerworks/community/alphaworks/tech/gaiandb, 2014. [Online; accessed 9-April-2014].