A Disk Based Stream Oriented Approach For Storing Big Data

Peter Membrey and Keith C.C. Chan
Department of Computing
Hong Kong Polytechnic University
Hong Kong SAR, China
Email: {cspmembrey,cskcchan}@comp.polyu.edu.hk

Yuri Demchenko
System and Network Engineering Group
University of Amsterdam
Amsterdam, The Netherlands
Email: y.demchenko@uva.nl

***Abstract*—This paper proposes an extension to the generally accepted definition of Big Data and from this extended definition proposes a specialized database design for storing high throughput data from low-latency sources. It discusses the challenges a financial company faces with regards to processing and storing data and how existing database technologies are unsuitable for this niche task. A prototype database called CakeDB is built using a stream oriented, disk based storage design and insert throughput tests are conducted to demonstrate how effectively such a design would handle high throughput data as per the use case.**

I. Introduction

Big Data has become one of the hottest topics in the computer science field. Although Big Data itself is not a new concept and many disciplines have been dealing with such data for decades, it is only relatively recently with the availability of cheap compute cycles and storage provided by Cloud Computing that the ability to process such large volumes of data has become practical for the majority of researchers.

However, although the focus is on processing Big Data in the cloud, there are still more traditional requirements for Big Data. Not all datasets are suitable for storage in the cloud due to technical (data size) and business (proprietary information) reasons. These systems require an on-site database that cannot take advantage of cloud architecture (i.e. the ability to scale out). This paper demonstrates that current database technology is not sufficient for this task and proposes an alternative database design specializing in storing low latency high throughput stream oriented data.

This paper examines a use case study from a financial company and its requirements for such a database. It highlights where the currently accepted definition of Big Data (the 3V model) does not fully capture the requirements of such a company. Extensions to the model that take Value into account are then proposed and discussed.

A prototype implementing the database design called CakeDB is tested for write performance and demonstrated to be significantly faster than MongoDB for the specified use case.

II. Definition of Value in Big Data

For data streams that run indefinitely, one of the biggest concerns for a company is the value of the data being stored. One of cloud Computing's defining features is the cost effectiveness of the resources that it offers. Despite lowering the cost of storage, such storage is still not free. Therefore when determining what data to store, the value of that data must be considered.

For example, market price data is a very specific form of data found in financial companies that contains the history of a given market or financial instrument. This data is used for back testing and verifying models and developing strategies amongst other things. The perceived value in having such data is that the data reflects how the market truly behaved at a given time and therefore allows more accurate simulations to be created. This level of data is not generally available for commercial purchase and so in order to maintain this data a company must record it.

However recording the data does not bring any value to the company. It only becomes valuable once that data is used or processed. With traditional amounts of data, storing almost anything indefinitely posed limited financial or technological challenges, but Big Data has changed this significantly. Now it is possible to fill vast amounts of storage space with data. The problem is that it is possible that the storage of such data greatly exceeds any potential benefit for storing such data.

Different database systems offer different cost benefits due to their design and structure as well as their intended use case. When considering Big Data it is crucial to examine the data being stored as well as the potential systems for storing that data. One system might be extremely expensive and hence unsustainable whilst another might offer similar features that can be sustained potentially indefinitely. Key to understanding this concern is being able to determine the likely throughput and hence total volume of data stream and then to determine whether the value of the data is high or low.

A. High Value Data

High Value Data (HVD) is data that has a known benefit from its storage. Companies that sell online storage space for example fall under this category. Theoretically, the more data that is stored, the greater the value will be. As long as the return on the data stored exceeds the cost of storing it, the company can continue to store such data indefinitely.

The value of data does not have to be measured as a dollar amount. However it needs to be understood that there is a

cost associated with data storage regardless of how that data is valued. Storing cultural or historical data has significant cultural value. The cost of storing that data might very well exceed the financial benefit drawn from storing it – but that is offset (and presumably paid for) by the community itself.

B. Low Value Data

Low Value Data (LVD) is data that is stored in the anticipation that value will be drawn from it in the future. Market tick data is a good example of this. Storing such data has limited use and generates no value when it is simply stored. When used for back testing and simulation, it is more valuable because it represents how a market did react to a given situation rather than how it theoretically should react. However this does not implicitly confer value to the data – similar results can be found from artificially constructed data as well. Such data becomes valuable when new systems, strategies and methodologies are developed based on that data. Until then the data has minimal value, but continues to be stored because of the potential for gaining value.

C. Storing the data

The key difference between LVD and HVD is the means of storage. With HVD, there is much less concern about storing the data because it generates an immediate or known return to cover such costs. LVD however may never bring a return on investment and so expensive storage systems are not likely to be used.

III. RELATED WORKS

Existing database technologies have become more focused on specific niche requirements. Michael Stonebraker and his fellow researchers are responsible for the H-Store [1] and C-Store [2] databases. However their database structures are relational and as such do not support highly structured 'NoSQL' datasets. The same is true for key-value store systems (such as memcached [3] and Riak), they were not designed to handle this sort of dataset.

There are two main choices for this style of data; MongoDB [4] and CouchDB [5]. CouchDB stores data in native JSON format and takes a unique approach to indexing data. Where RDMS systems have static data and dynamic queries, CouchDB implements dynamic data and static queries [6] leading to fast reads only when indexes are available. CouchDB also has a REST (REpresentational State Transfer) interface which makes querying less efficient. Insert performance, even with batch jobs is relatively, around 700 updates per second.

In contrast, MongoDB is more like traditional SQL databases. It is possible to do dynamic queries and due to its BSON (Binary JSON) storage format offers JSON-like structures but with very fast search and parse times. However being memory mapped, physical memory is a highly limiting factor. Once data exceeds this limit, performance drops to the point of being unusable.

These limitations make such databases unusable for handling Low Latency High Volume structured data streams. Structured data streams need to be able to store data in a JSON-like format, ruling out RDMS and key-value storage systems. The system also needs to support high throughput writes of at least 25,000 inserts per second. With a REST interface, CouchDB cannot support this demand, leaving MongoDB as the only viable choice. However being memory based the amounts of storage is greatly limited, particularly for HVLV data streams.

IV. USE CASE – FINANCIAL TRADING COMPANY

One of the industries most affected by Big Data is the financial sector. Businesses that operate in this sector must manage a large variety of different data requirements, all of which are core to the success of a business. This case study will examine the different sources of data and the data structure.

There are two fundamental sources of data for a financial company. Data can be classified as either internal or external data depending on the source and destination of the data. Generally, external data and internal data are not directly compatible and must be converted to and from the relevant data structure when the data crosses the company boundary. This means that a company must be able to process each of the external data formats into an internal form that its systems can handle as well as being able to convert it back to an external format when required. Figure 1 highlights how different systems may inter-operate in a given trading system.

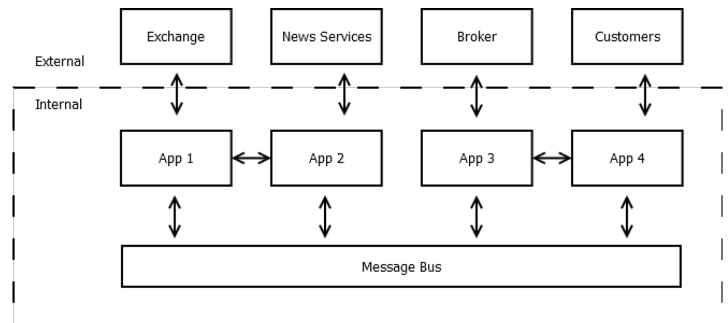

Fig. 1. Trading System Message Flow

Financial companies need to store extensive amounts of highly structured data for extended periods of time. This data is generally highly structured and whose structure is in a constant state of flux. Over time many terabytes of data will easily be accumulated. Data needs to remain quickly accessible while maintaining low latency high throughput. Specifically, the impact on a sending application should be minimized to prevent delays in the database from introducing latency and response spikes in the application itself.

V. DESIGN REQUIREMENTS

There are two key design requirements for such a database. First, data needs to be inserted deterministically with respect to the client and with minimum impact. Second, the data should be easily extractable regardless of the size of the database. Queries should run in a reasonable amount of time, but high performance is not a requirement. Thus while it is not acceptable for the database to cause a client application to take a performance hit, it is acceptable for read queries to take a performance hit.

These design requirements break down further as follows:

- Does not block the client
- Has low impact on the client
- Can accept data at high speed
- Takes full advantage of available resources
- Copes with burst traffic
- Can operate on a single machine
- Queries are executed in a reasonable time

CakeDB was designed with each of these requirements in mind. It has a simple binary streaming interface for receiving inserts and due to its internal architecture can never block the client (barring any TCP buffering issues etc). It can accept data at high speed and, using the Erlang Open Telecommunications Platform, is able to fully utilize all resources on a machine (particularly multi-core). It uses memory to buffer incoming requests and a latching mechanism to take the actual writing process off the critical path. This allows it to handle large amounts of burst traffic and to do so on a single machine.

It was also designed to be used as a fast storage engine so that client applications writing extensive amounts of data would not be impacted by any slowness in the storage layer. Relational databases have to maintain multiple different views of data and indexes need to be calculated and data processed. This approach makes inserts non-deterministic and therefore not ideal for low latency real time environments. MongoDB writes data directly to memory, using memory mapped files as back-end storage. This combined with 'unsafe writes', allows MongoDB to have extremely fast insert performance. However it must still convert the data payload to BSON for storage and update any relevant indexes. Therefore while MongoDB is far more deterministic than a relational database, the overhead involved in processing and storing the data can be extensive.

For executing queries CakeDB currently has very limited features. As a natural order database, CakeDB can either return all data in a particular time range or all data since a specific time. This allows chunks of relevant data to be quickly retrieved and loaded into more traditional analysis tools. The ability to request 'all changes since' allows 'Sweeper' applications to be written, polling the database at regular intervals and retrieve all of the new results for inserting into other databases. This allows CakeDB to operate as both a long term storage system (ideal for when datasets exceed the threshold for a given processing database) as well as a buffer system to move slow processing tasks off the critical path (useful for latency sensitive applications).

CakeDB is highly specialized and would be of limited use as a general purpose database. However for handling high speed streaming data, it is able to consistently out perform the alternatives that are currently available. The following sections discuss these design requirements in greater detail.

A. *Does not block the client*

Particularly when dealing with this type of data, it is critical that the capturing application cannot be blocked by the database. An insert might normally take 100 microseconds but should the database decide to write an index or handle a query from elsewhere, this could block an insert which might take 1000us instead, potentially creating an unacceptable backlog and delays elsewhere in the system. For example, it would be unacceptable for a market feed capturing tool to slow down as this could lead to severe financial consequences.

A database that does not block the client must operate asynchronously and there should not be a response cycle where each insert is acknowledged before the next insert can be sent. The database must accept a stream of data from the client and continue to do so. If the database cannot handle the load, it should drop the connection, an error condition that the client can easily handle. This would prevent the client application from creating a backlog of data, overwhelming the program.

B. *Has low impact on the client*

When thousands of messages are being processed each second, even small amounts of processing overhead per message can greatly affect a system's performance. The time needed to convert native data into a format the database driver can understand can be significant. A database that can handle high throughput data needs to limit the effect of using that database on the client. Ideally sending data should be as close to sending the native format on the wire as possible. This means the database design needs to be aligned for the sorts of data that will be sent.

It also means that as much of the processing logic should be pushed to the server side as possible, making the database driver a basic socket wrapper rather than be responsible for data translation. This in turn means that the database must handle that task instead. By aligning the data structures in the database to those sent by the client the need for heavy data processing is reduced although not eliminated entirely. With clients generally having more CPU power available to them, the shift to pushing the load on to the database server opposes the current trend of near-side processing. However CPU power alone is not enough to resolve the latency issues (throughput then becomes a problem) and many newer collection devices are low-powered embedded systems or related devices. Processor cycles on such devices are at a premium and thus it is desirable to push the workload on to the server.

By aligning the database with the data that it needs to store (potentially with multiple data storage types) and pushing the processing on to the server, the impact on the client for sending data to CakeDB is greatly reduced. Combined with guaranteeing that the client will never block and that there is no need for extensive error checking code, means that the footprint on an application for using CakeDB is little larger than using raw TCP sockets directly.

C. *Can accept data at high speed*

High speed is defined as 10,000 updates per second. This is reasonable based on known flow rates on financial markets and anticipated flow rates from services such as Twitter. Many databases can easily burst to this level of inserts but depending on the database, maintaining a steady stream of data at this speed can push some beyond their limits.

In order to be able to handle high speed data, the database must limit the amount of processing or overhead that occurs

during the writing process. As 10,000 updates per second as the minimum benchmark, if the database cannot clear this many updates per second, it will ultimately have a backlog from which it can never recover. This would slow down the database and has the potentially to ultimately cause it to fail.

D. Takes full advantage of available resources

Due to the type of data being stored (low value, large storage), being able to take full advantage of a given machine's resources is paramount. For these use cases, it is not possible to solve the problem by simply adding more hardware as the collection of more data does not necessarily bring with it an expected increase in profitability. Thus whilst a reasonable specification machine may be provided for the data collection (or perhaps several), the number of machines will not be able to grow much beyond their initial number.

To make the most of available hardware, a database should make full use of all available cores on a given machine or cluster. The database should be able to maximize it's throughput without requiring special configuration or optimization at the code level for different platforms. The Erlang OTP (Open Telecommunications Platform) emphasizes a very powerful process based application development model which allows the Erlang VM to fully utilize all CPU cores in an efficient way and was designed for the high throughput and soft real time requirements of the telecommunications industry.

E. Copes with burst traffic

Although the benchmark is 10,000 messages per second, it is possible for such feeds to burst well beyond this to 100,000 messages per second or more. The database must be able to cope with periods of intense bursting and recover without impacting the client. This is challenging because a burst of a factor of 10, would put the database under equally higher load which it may not be able to handle. Performance degradation can happen during these burst periods leading to residual performance issues at regular speeds once the feed returns to normal.

To properly handle burst traffic a mechanism is needed that can buffer or absorb the load for a period of time. This can be done by using RAM to buffer data for example. Whatever solution is used, the sudden burst should not have any noticeable effects to the client application.

F. Can operate on a single machine

Related to the previous section on efficient resource usage, the database needs to be able to operate appropriately on a single machine of sufficient specification. That is, the problem should not be solved by creating a large database cluster as this amounts to using brute force. There will undoubtedly be feeds that a slower machine could not handle but a faster machine could and this comes down to specifying the right machine for the task. However that is different from using substantial resources to allow an inefficient system to still perform at a reasonable and acceptable level.

Using Erlang and not requiring large amounts of memory, the database will be able to take full advantage of the hardware and to use it efficiently. As the database is specifically designed for high throughput workloads, the design reflects trade offs that a general purpose database cannot make and thus is able to perform at a much higher level on comparable hardware.

G. Queries are executed in a reasonable time

CakeDB was designed with a focus on high speed, low impact inserts. However retrieving data from CakeDB is also important. Querying data from a heavy over loaded database is extremely slow, where simple count queries can take many minutes to execute. This is the case with MongoDB when the dataset and indexes greatly exceeds the amount of physical memory in the machine. MongoDB has no concept of disk access and will access memory as it sees fit. This causes a huge number of page faults, as MongoDB tries to access the index (and hits disk), tries to access the data (and hits disk again), followed by another index read (which will likely hit the disk).

This level of performance is unacceptable but it is clearly caused by using the database outside of its designed parameters. CakeDB must respond to queries within a reasonable time frame. However CakeDB provides querying tools suitable for data mining and extracting data for further processing. It is not designed for OLAP or OLTP. It provides a minimal interface for extracting blocks of stored data from streams but currently does not have enhanced querying capability, a feature that is highlighted under future work.

VI. Natural Order Storage

CakeDB is a natural order stream based database. This means that data is stored in the order it is received by CakeDB and assigned a unique timestamp as an identifier. The timestamp used is guaranteed by the VM to monotonically increment and thus not only describes each document stored but also its order. This is only effective on single machine instances and so for clustered operation some form of additional identifier will need to be derived.

Due to the nature of the data, data-files are written to in an append-only fashion. This provides some protection against data corruption but also makes it easy to reason about the location of a given data item. In effect the stream is indexed based on insert time and thus data extraction with a time range is extremely fast and efficient. To improve performance further a simple indexing system is used where the byte location of a data item is recorded with a timestamp every 1,000 documents or 50MB of data inserted. It is expected that this option could be tunable based on the streams use case.

A. Binary Format – Disk

CakeDB also stores data in a binary format on disk. The current scheme is simplistic and not suitable for production usage. This is being actively developed to add greater robustness to the system. Data is stored with a 16 byte header consisting of the document size, its timestamp and one of several options (currently not used). A secondary index file is used to allow fast searching to any data point in the stream.

B. Binary Protocol – Wire

For performance reasons, CakeDB implements a binary protocol. The protocol has a set 6 byte header consisting of a 4 byte message length and a 2 byte integer specifying the command. All data is encoded in big-endian format for use on the wire.

The format is very efficient for sending data for most languages and does not require any supporting code or additional parsers to prepare the data for sending. All processing is effectively pushed to the database server. CakeDB currently only supports a raw stream and so it is highly efficient for storing data regardless of the composition. With a raw stream, clients can store POJO or other arbitrary binary data without any trouble.

VII. Streaming inserts

A database supports streaming inserts when it does not need to acknowledge receipt of data before the client can send additional data. In the general case it is expected that the database is able to handle the data that is sent to it. Should the database fail, the client should be notified (in the case of CakeDB this is done via a disconnect). There is limited benefit in high speed systems of this kind to send individual status messages. Often by the time the client is able to recognize that an error condition has occurred, it will no longer have the data it needs to correct the situation. Assuming that the client did have the data the delays incurred trying to recover from the error condition could be significant and would potentially be unrecoverable and make a bad situation worse.

CakeDB is designed on the "let it crash" paradigm [7], i.e. fail fast and fail early. This requires that as soon as an error condition is detected, the process or application should fail immediately. With CakeDB, as soon as an insert error is detected, the client is immediately disconnected allowing the client to recovery quickly by reconnecting and continue sending data. This form of failure is extreme in that it would cause data loss. However it also limits the amount of data that would be lost and prevents potential cascade failures as multiple applications try to handle the data loss and start having issues of their own.

CakeDB does not acknowledge inserts with the implied contract, that if the client is not disconnected, CakeDB will ensure the data is persisted to disk. This in turn means that the client can (and should) send data as fast as it is able to.

VIII. Adopting the Erlang OTP

Erlang was developed by a small team at Ericsson lead by Joe Armstrong with the first version being completed in 1986 [8] and ultimately open sourced in 1998. Erlang was developed to address a key issue in software engineering, that of stability and reliability in systems known to be susceptible to errors [9]. Armstrong believed that existing multi-threaded share-everything designs were too inter-connected to allow faults to be properly isolated and believed that a process based messaging passing model would be sufficient [10]. This design also makes Erlang exceptionally good at utilizing multi-core systems as the process model prohibits shared state and hence there is extremely little locking in Erlang [11].

The message passing paradigm is not unique to Erlang and can be found in other languages such as Scala written for the JVM (Java Virtual Machine) [12]. However the challenges required for integrating a message based paradigm with a thread based paradigm are not insignificant [13]. Although Scala allows both paradigms to be used simultaneously, ultimately it must execute code on the JVM. As this is inherently shared-state and thread based, there are limits to what a language such as Scala can do. Erlang on the other hand has a VM written in C that is designed from the ground up to fully support the message passing paradigm [14]. This combined with Erlang's included libraries for easy integration with C and Java [15], makes Erlang a very strong candidate for CakeDB.

In Erlang, each task is represented by a lightweight process. These processes communicate with each other by sending and receiving messages. Each process is independent in its own right and does not share state with any other process. This provides a unique form of isolation that allows processes to fail without necessarily affecting any other process (unless the developer so wishes). Erlang takes advantage of this unique form of isolation and combines it with a supervisory tree that allows processes to inform other processes that they have failed (or simply exited). This allows other processes (generally specifically designed for the task) to restart the failed process or take any corrective steps. When written properly, an Erlang application is extremely resilient to failure.

This is desirable when writing a database but is not the prime benefit for CakeDB. By isolating application logic into processes, there is no shared state and internal state is kept within the process itself. Processes operate on messages that they receive. This means that as long as the process has messages, it can be processed completely independently from any other process. This benefit is easily seen on a multi-core system where there may be 12 or more cores. Erlang is able to schedule work across all 12 cores efficiently as there are no inter-dependencies. For CakeDB, each stream is unique, it does not need to share information with any other stream. This allows for easy parallelism across streams.

IX. CakeDB Latched writer

CakeDB uses a latched writer to ensure minimal impact on client performance. Each client connection has a dedicated Erlang process that handles all of that processes communication with CakeDB. To ensure minimal impact when writing to CakeDB, it is essential that this client process does not have to make any blocking calls. Specifically it must not be responsible for any tasks that could effectively block the process from handling incoming data from the socket.

To achieve this, the client process takes the data and passes it to the process responsible for handling that particular stream. This decouples the client handling process from the writing process and ensures that it cannot be blocked. As the stream process can be receiving very large amounts of data at any given time, it is highly likely that incoming message load may exceed the machines capacity for writing the data to disk. To ensure that the stream process does not become disk bound, a second process is responsible for handling disk I/O. When there is data to be written, the stream process will send the bundled data (the stream process is responsible for processing the data ready for storage) to the writer process.

At this point, the stream process will start to buffer newly arriving data internally. When the writer process has committed the data to disk, it will send a 'clear to send' message to the stream process. When this message is received, the stream process will transfer the bundle to the writer process and return to buffering data. This design ensures that as long as there is sufficient ram in the system to handle the load bursts, CakeDB can continue to provide high performance, even under very heavy load.

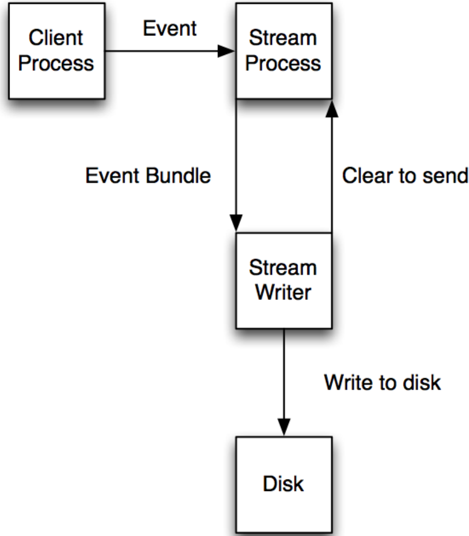

Fig. 2. Latched Writer

X. CakeDB Internal Structure

CakeDB is built on the Erlang OTP platform and as such has a process orientated design and follows best practice as outlined by Joe Armstrong [16]. Each of the core components of CakeDB are implemented as independent functions. These functions communicate via message passing. There is no shared memory and no shared state as is common in most concurrent systems. By avoiding these (mutexes and related locks), Erlang (and hence CakeDB) can avoid the heavy concurrency related penalties paid by other systems (such as MongoDB's global write lock) and gain considerable performance benefits.

XI. Internal Processes

CakeDB uses Range to provide a TCP acceptor pool. Range creates a number of processes that wait for an incoming TCP connection to handle. Range defines the process structure of a protocol which is then implemented by the developer. This is where the application logic for handling the connection is placed.

A. Stream Manager

The Stream Manager (SM) is responsible for initiating streams and assigning them a unique Stream ID (SID). During communication with a client, all read and write requests must contain the SID of the stream that the client wishes to write to or read from. The SM is currently very simplistic and does not maintain state. Future implementations could add additional features such as storing stream statistics, access credentials and other related services. The flow of the process can be seen in Figure 3.

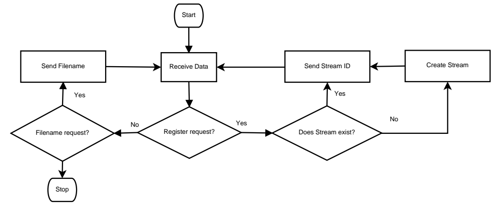

Fig. 3. Stream Manager Process Flow Chart

B. Stream Process

The Stream Process is responsible for receiving write requests from the connection processes. Once a write request is received it is added to an internal buffer in the process. The Stream Process works in tandem with the Stream Writer Process (SWP). When it receives a Clear To Send (CTS) message from the SWP, it will send the buffer to it. It then continues to buffer data until the SWP sends another CTS message (see Figure 4).

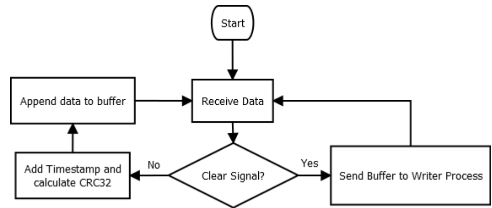

Fig. 4. Stream Process Flow Chart

The Stream Process initially starts with *ClearToSend* set to *true*. This means that the first time it receives a data message it will be sent to the SWP rather than held in a buffer. After this initial send, *ClearToSend* is set to *false* and all future data messages will be added to the buffer after processing. The buffer will only be sent to the SWP after another CTS message has been received. In this way, there is a feedback mechanism between the process receiving the data (SP) and the process handling the file IO (SWP). As this is an asynchronous relationship, delays in file IO will not impact the SP.

Data processing is kept to a minimum to avoid any unnecessary overhead. The CRC32 checksum is calculated for the message and the headers (defined in the file format) are added.

The Stream Process also implements a 5 second timeout when waiting for new messages. This timeout allows the process to do house keeping such as executing the garbage collector and checking to see if any data needs to be sent to the SWP. There is also a 50,000 message maximum buffer size. When this is reached, the SP will automatically forward the buffer to the SWP regardless. This is a failsafe device that should never be needed in practice.

C. Stream Writer Process

The Stream Writer Process (SWP) is responsible for writing data to disk as well as updating the index file. When first

initialized, the SWP checks its statistics to determine whether it needs to update the index file. Initially it will have written no data, so this section will be skipped. At present the current implementation updates the index after every 50MB or 1,000 messages are written to disk. It then calls *receive* to wait for a new message from the Stream Process (SP). Once a new message is received, the SWP will write the data to the file and then send the *Clear To Send* (CTS) message back to the SP. It then repeats the index checks before waiting for the next message (see Figure 5).

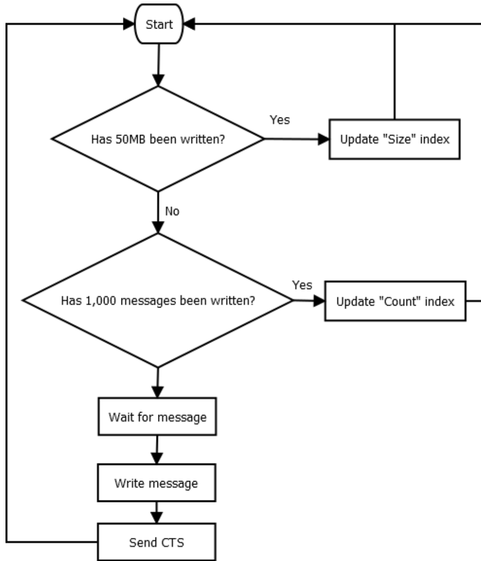

Fig. 5. Stream Writer Process Flow Chart

The SWP also implements a 5 second timeout which is used to manually trigger garbage collection.

D. File Manager Process

The File Manager Process is part of the Erlang VM. In order to improve efficiency (and remove some of the concurrency pitfalls when using native IO) all read and write requests go through a special Erlang process. This process then handles the IO on behalf of the Erlang VM.

It is possible to assign special 'asynchronous threads' to the Erlang VM. These threads are used specifically for IO operations so that the Erlang scheduler threads do not block waiting for IO to be handled.

XII. File Format

CakeDB currently has a very basic file format. The initial file format was very rudimentary consisting solely of a timestamp, length and payload. This performed very well but should the database fail or data be damaged, there would be no easy way to detect the failure. Therefore a more robust file structure for the data file was devised (see Section XII-A). In addition to the data file, CakeDB also generates an index file. This file is still very basic consisting of a timestamp, a type and a byte offset. There is currently no redundancy or headers for this file (see Section XII-A).

Timestamp 64 bit integer	Index Type 8 bit integer	Byte Offset 64 bit integer

Fig. 7. CakeDB Format - Index File

A. Data File Format

The data file format is designed to be optimal for reading sequentially. Due to CakeDB's natural order design, requests for a block of data (that is from timestamp A to timestamp B) can be retrieved very efficiently. The structure is show in Figure 6.

SoM 3 bytes	Timestamp 64 bit integer	Message Size 32 bit integer	CRC32 32 bit integer	SoP 3 bytes
Binary Payload X bytes				EoM 3 bytes

Fig. 6. CakeDB Format - Data File

SoM (Start of Message), SoP (Start of Payload) and EoM (End of Message) are all three byte repeated sequences. They are used by CakeDB to determine whether the message read from disk follows the correct structure. This correctness is tested using Erlang's binary pattern matching. If these sequences are not in the expected locations, CakeDB will throw an error. It is anticipated that this mechanism can also be used to recover data files that become corrupted as CakeDB will be able to determine the last valid message and remove the damaged section.

B. Index File Format

The current implementation of the Index File Format is simplistic. It does not implement any error checking and is effectively an array of timestamps and byte offsets. This file is iterated when executing a query to find the closest offset for a given timestamp. This is then used to open the data file at the right location. Although there has been no optimization done on the index file (it is opened and read sequentially for each query in the current implementation), it remains extremely small compared to the data file size. A data file of 250GB could have an index file only 100KB in size. Sequentially reading a binary file has limited impact on performance.

The structure of the index file can be seen in Figure 7. The *Type* field is currently unused by the query engine but allows the process updating the index to specify what action caused the index update. For example, were 1,000 messages written or was the 50MB limit reached?

The index file could be improved in a number of ways such as by adding redundancy and proper headers. Further performance could probably be gained from moving this data to an in-memory data structure. However given the current performance requirements, this has yet to be a concern.

XIII. Initial Throughput Experiment

A benchmarking script took a fixed JSON message and then attempted to store that message using the most appropriate method for the given database. There were two messages used

for this test. The first was approximately 3KB in size (3,228 bytes) and the second was approximately 1KB in size (1,158 bytes). For each test, a different number of messages was inserted, starting at 1,000 and doubling each time to reach a total of 256,000 messages. Each test was run 5 times and an average taken to give the final time. Using this time and the number of messages inserted, the total throughput of messages per second was calculated.

A. 1KB document size

Based on the average value from each run, CakeDB achieved a maximum throughput of *64,981* (Table I) inserts per second, compared with MongoDB which achieved *7,664* (Table II), making CakeDB capable of approximately *8.5* times the throughput of MongoDB.

TABLE I. TIME TAKEN TO INSERT 1KB DOCUMENTS INTO CAKEDB

Number of inserts	1,000	64,000	256,000
Average time taken (seconds)	0.02	1.11	4.60
Throughput (inserts per second)	64,981	57,726	55,592
STDEV	0.003	0.050	0.141

TABLE II. TIME TAKEN TO INSERT 1KB DOCUMENTS INTO MONGODB

Number of inserts	1,000	64,000	256,000
Average time taken (seconds)	0.13	8.86	33.68
Throughput (inserts per second)	7,664	7,221	7,602
STDEV	0.002	0.636	1.205

B. 3KB Document Size

A fully populated Twitter message is around 3KB in size. As a capture tool connected to a full Twitter feed (the Firehose Streaming API) would need to handle this size of document, a 3KB document test was set up to simulate a worst case scenario. The tests were repeated as per the 1KB document test. However the results were similar in scale despite the change in size. CakeDB achieved a top throughput of *51,266* (Table III) and MongoDB of *4,391* (Table IV).

TABLE III. TIME TAKEN TO INSERT 3KB DOCUMENTS INTO CAKEDB

Number of inserts	1,000	64,000	256,000
Average time taken (seconds)	0.02	1.26	5.23
Throughput (inserts per second)	51,266	50,711	48,956
STDEV	0.003	0.018	0.173

TABLE IV. TIME TAKEN TO INSERT 3KB DOCUMENTS INTO MONGODB

Number of inserts	1,000	64,000	256,000
Average time taken (seconds)	0.23	14.72	55.87
Throughput (inserts per second)	4,391	4,347	4,502
STDEV	0.011	0.442	0.512

Both databases dropped in throughput with the larger message size. Although the message size was approximately three times large, CakeDB lost 21.1% in throughput whereas MongoDB lost 42.7%.

C. Pre-hashed JSON

The simulation remains as close to real world usage as possible by requiring both databases and their drivers to handle the incoming JSON data. For CakeDB this does not pose a problem because its standard stream type is raw, that is, it store data in the same format that it is received. This means it does not need to process the JSON data into an intermediary data format or do any processing on the message itself.

MongoDB however must convert the data to BSON, a binary based implementation of JSON. This first requires parsing the JSON into Ruby's native hash structure and from there converting it to BSON data. JSON parsing would be required in any application performing this task and so as a real world test, it is fair to include this load during MongoDB's tests. The YAJL C library is used in these tests which is currently considered to be one of the most efficient implementations available. The Ruby MongoDB driver is also using the native C BSON driver, so the use of Ruby is not unfairly affecting the results.

However to determine whether the performance issues are in the JSON library, an additional test was done where the data was converted to a hash only once. This hash was then passed to the MongoDB driver. This was repeated for both document sizes and get the results in table V and VI for both the 1KB and 3KB tests respectively.

TABLE V. TIME TAKEN TO INSERT 1KB DOCUMENTS INTO MONGODB PRE-HASHED

Number of inserts	1,000	64,000	256,000
Average time taken (seconds)	0.08	4.79	19.12
Throughput (inserts per second)	12,046	13,360	13,387
STDEV	0.004	0.060	0.114

TABLE VI. TIME TAKEN TO INSERT 3KB DOCUMENTS INTO MONGODB PRE-HASHED

Number of inserts	1,000	64,000	256,000
Average time taken (seconds)	0.11	6.54	25.36
Throughput (inserts per second)	9,257	9,784	10,095
STDEV	0.005	0.427	0.792

These tests demonstrate that a significant amount of the time is being spent parsing the JSON to a Ruby hash. With the 1KB documents, performance increased by approximately 38.4% and by 52.6% for the 3KB documents. This suggests that as the complexity and size of the document increases, the time required to parse it also increases. However despite removing the JSON parsing from the test, CakeDB remained 5.4 times faster in the 1KB test and 5.5 times faster in the 3KB test.

D. Conclusions

The initial tests with both 1KB and 3KB documents showed that CakeDB has better throughput for handling inserts. This is due to its low impact client, streaming inserts and latched insert handling on the database itself. The results demonstrate that a specialized database can out perform a general purpose database and that CakeDB has significant performance benefits over MongoDB. Additional tests showed that even with pre-hashed datasets and using bulk inserts, MongoDB was not able to approach the performance offered

by CakeDB. This suggests that the performance difference is not simply due to external overhead (such as JSON parsing).

XIV. Summary

Currently in industry there are numerous workloads that do not lend themselves well to a 'scale out' approach. This paper proposed a disk based approach for storing and then retrieving sequential data. By using disk rather than RAM as the main form of storage, this approach is able to store significantly more data on a given machine.

XV. Future Developments

Currently CakeDB only supports simple time range based queries. This works well for reporting, but limits its applicability to more general use cases. To address this, support is being added for BSON (Binary JSON) as used in MongoDB along with a SQL like querying language. This will allow for richer and more precise queries.

Although CakeDB did not initially focus on read performance, in general usage, read performance was significantly better than expected. Further research will be undertaken to determine whether or not such an approach could yield similar read performance to current systems (such as MongoDB) when accessing sequentially stored data.

Including value as part of the definition for Big Data raises questions as to how value can or should be defined. It could be said that the value of data consists of both intrinsic and extrinsic value. Further research will be conducted to see whether pricing algorithms (such as those used in pricing financial derivatives) could be used to provide a value benchmark.

References

[1] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang *et al.*, "H-store: a high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.

[2] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil *et al.*, "C-store: a column-oriented dbms," in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 553–564.

[3] J. Petrovic, "Using memcached for data distribution in industrial environment," in *Systems, 2008. ICONS 08. Third International Conference on*. IEEE, 2008, pp. 368–372.

[4] P. Membrey, E. Plugge, and T. Hawkins, *Definitive Guide to MongoDB*. Apress, 2010.

[5] J. Lennon, *Beginning CouchDB*. Apress, 2009.

[6] J. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide: Time to Relax*. O'Reilly Media, 2010.

[7] S. Vinoski, "Concurrency and message passing in erlang," 2012.

[8] F. Cesarini and S. Thompson, *Erlang programming: A concurrent approach to software development*. O'Reilly Media, Incorporated, 2009.

[9] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, KTH, 2003.

[10] ——, "How erlang views the world and what we have learned in the last 25 years of programming distributed systems." *EPTCS 58*, 2011.

[11] ——, "erlang," *Communications of the ACM*, vol. 53, no. 9, pp. 68–75, 2010.

[12] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," Technical Report IC/2004/64, EPFL Lausanne, Switzerland, Tech. Rep., 2004.

[13] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2, pp. 202–220, 2009.

[14] B. Hausman, "Turbo erlang: Approaching the speed of c," *Implementations of logic programming systems*, pp. 119–135, 1994.

[15] J. Larson, "Erlang for concurrent programming," *Communications of the ACM*, vol. 52, no. 3, pp. 48–56, 2009.

[16] J. Armstrong, "Programming Erlang: Software for a Concurrent World," 2007.