Introduction

This report describes a series of experiments to evaluate technologies suitable for developing a real-time alert processing platform for LSST:UK capable of handling the data rates expected during the lifetime of the LSST project.

The evaluation criteria for the experiments are based on the following requirements:

- Capable of meeting the initial expected data rate from the LSST project.
- The ability to increase the processing speed of the system by adding new resources.
- The ability to increase the storage capacity of the system by adding new resources.
- How much downtime, if any, is required to add new resources to the system.
- The resilience of the system to individual component failures.

An additional evaluation criteria is set by the requirements of the LSST:UK project and its links to the IRIS eInfrastructure initiative and the DiRAC integrated supercomputing facility.

- Where possible implement the system using standard hardware resources.

The target data rates for the experiments are:

- Sustained 1,000 alerts per second
- Stretch goal 10,000 alerts per second

These numbers are based on the values given in table 29 of the LSST System Science Requirements Document.

The LSST System Science Requirements Document (LPM-17)
https://docushare.lsstcorp.org/docushare/dsweb/Get/LPM-17/LPM17_LSSTSRD_20180130.pdf

> *"Specification: The system should be capable of reporting such data for at least transN candidate transients per field of view and visit (Table29)"*

Quantity	*DesignSpec*	*MinimumSpec*	*StretchGoal*
transN	10^4	10^3	10^5

Table 29: The minimum number of candidate transients per field of view that the system can report in realtime

The *transN* value sets the number of alerts per visit, this combined with the expected cadence of 30 to 40 seconds per visit gives us our target evaluation criteria of 1,000 alerts per second and the stretch goal of 10,000 alerts per second for these experiment.

To meet the scalability and fault tolerance requirements the architecture design for the alert processing system is based on a distributed micro-service architecture, where multiple instances of each component can be deployed in parallel, using Kafka data streams to distribute the alert data between each stage of the pipeline.

The key advantages of using a distributed micro-service architecture are :

- It provides a fault tolerant system with no single critical point of failure.
- It makes it easier to scale the processing and storage capacity in response to changes in system load.
- It makes it easier to modify or replace components in response to changes in science requirements.

Within the scope of these experiments the software for each experiment is implemented as a prototype implementation of the component along with one or more simple JUnit tests that demonstrate that the component is able to meet the target criteria.

Crossmatch algorithms

The following experiments compare the Hierarchical Triangular Mesh (HTM) algorithm used to index many of the WFAU catalogs with the zones based algorithm described in a 2004 paper by *J. Gray et al.* to identify the best option for implementing a cross match component capable of matching the live alert stream against one or more of the large science catalogs held by WFAU.

The target criteria for these experiments is to demonstrate a cross match implementation capable of meeting the target of 1,000 alerts per second, and be able to scale up to meet the higher stretch goals by adding additional resources.

The source code for these tests is available on GitHub :
https://github.com/lsst-uk/enteucha

Hierarchical Triangular Mesh (HTM)

The Hierarchical Triangular Mesh (HTM) algorithm starts by dividing the celestial sphere into 8 spherical triangles, and then builds a quad-tree recursively decomposing each triangle into 4 sub-triangles.

See the following references for details of the algorithm :

The Hierarchical Triangular Mesh, P. Z. Kunszt et al.
https://link.springer.com/chapter/10.1007/10849171_83

The Indexing of the SDSS Science Archive, P. Z. Kunszt et al.
http://www.skyserver.org/HTM/doc/adass99.ps

SkyServer HTM Documentation
http://www.skyserver.org/HTM/doc/intro.aspx

The code to implement the HTM algorithm in our tests are based on the Java library available from the SkyServer website at John Hopkins University.

http://www.skyserver.org/htm/implementation.aspx#download

Due to licensing issues we are unable to make this part of our code open source at this time.

Zone based algorithm

The zones based indexing uses a much simpler method of dividing the celestial sphere into thin horizontal zones based simply on the position declication.

```
zoneNumber = floor((dec+90) /zoneHeight)
```

See the following paper for details of the algorithm :

There Goes the Neighborhood: Relational Algebra for Spatial Data Search, J. Gray et al
https://arxiv.org/pdf/cs/0408031.pdf

More recently, the same zones based indexing has been used as part of the Astronomy eXtensions for Spark (AXS) Spark extension and library.

AXS: A framework for fast astronomical data processing based on Apache Spark
https://arxiv.org/abs/1905.09034

The initial steps of the zones based algorithm are much simpler to implement and faster to execute than the equivalent steps of in HTM algorithm, requiring only simple floating point and integer arithmetic operations compared to the complex trigonometry involved in the HTM algorithm. The trade off is to start the selection process using very simple steps, significantly reducing the amount of data involved, and then use more complex steps later in the process.

Database platform

Performing crossmatches on large datasets in conventional database systems is often limited by I/O performance. The overhead of getting the data off disc can obscure the relative performance of the indexing algorithms.

In order to maximize the performance, the following experiments were performed using in-memory databases. Once the I/O data access bottle neck is removed from the equation, the differences between the indexing algorithms becomes much more significant.

The database tests used an in-memory instance of the HSQLDB database engine to evaluate the algorithms.

HSQLDB (Hyper SQL Database) Java relational database management system.
http://hsqldb.org/

HTM vs zone comparison

The HTM tests used the Java library from JHU to calculate which HTM triangles intersected the target region and then queried the database to find all the sources within those triangles.

The zone based tests calculated which zones intersected the target region and then queried the database to find all the sources within those zones.

The tests showed a significant advantage to the zone based algorithm, demonstrating a performance increase of a factor of 10 compared to the HTM algorithm.

- 4,004,000 rows of test data
- HTM algorithm : found [10] in [213]ms
- Zone algorithm : found [11] in [17]ms

It is important to note that development time for this project was limited, and we did not set out to perform an accurate performance benchmark of the algorithms.

The objective of these experiments was to check the findings in the paper by Gray et al, that the zone based algorithm was simpler to implement and performed significantly faster than the HTM algorithm, and then move on to exploring how fast we could get the zone based algorithm to work.

As a result, the database queries and indexing used in the HTM version were not optimized. If an

accurate benchmark is needed then it would be worth re-visiting the code and optimizing the HTM implementation to get the best performance from it.

Database indexing

The next set of tests looked at the database query and indexing used in the zone based algorithm.

The SQL query used for the tests came from the query outlined in the paper by Gray, which included all three steps in the same query. The initial selection for the target Zone based on declination, the selection within the zone based on right ascension, and then a final selection based on distance.

```
SELECT
    ...
 FROM
    zones
 WHERE
    zone BETWEEN ? AND ?
 AND
    ra BETWEEN ? AND ?
 AND
    dec BETWEEN ? AND ?
AND
    (power((cx - ?), 2) + power((cy - ?), 2) + power(cz - ?, 2)) < ?
```

The tests evaluated three different indexing schemes, the first scheme created three separate indexes, one index on the integer zone id, one on the right ascension and one on the declination.

```
CREATE INDEX zoneindex ON zones (zone)
CREATE INDEX raindex   ON zones (ra)
CREATE INDEX decindex  ON zones (dec)
```

The second scheme created two separate indexes, one index for the integer zone id alone, and one index on the right ascension and declination combined.

```
CREATE INDEX zoneindex  ON zones (zone)
CREATE INDEX radecindex ON zones (ra, dec)
```

The third scheme created a complex index of zone id, right ascension and declination combined.

```
CREATE INDEX complexindex ON zones (zone, ra, dec)
```

The database tests showed that for this particular complex query, the combined and complex indexes performed better than the separate single value indexes.

- 2,563,201 rows of test data
- Separate [zone], [ra] and [dec] indexes : search time 83ms
- Ccombined [ra + dec] index : search time 50ms
- Complex [zone + ra + dec] index : search time 38ms

CQEngine implementation

At this point we began to develop a native Java implementation of the zone algorithm using the CQEngine library to index data in Java Collections.

CQEngine - Collection Query Engine
https://github.com/npgall/cqengine

This version implemented the zone algorithm directly in Java code, using the CQEngine classes to implement an indexed collection of zones:

```
public class ZoneMatcherImpl
implements ZoneMatcher
    {
    ....
    private final IndexedCollection<ZoneImpl> zones =
        new ConcurrentIndexedCollection<ZoneImpl>();
    ....
    }
```

and an indexed collection of positions within each zone:

```
public class ZoneImpl
implements Zone
    {
    ....
    private final IndexedCollection<PositionImpl> positions =
        new ConcurrentIndexedCollection<PositionImpl>();
    ....
    }
```

These tests showed a significant advantage to the native Java implementation, which out performed the HSQLDB database implementation by a factor of 10.

- 2,563,201 rows of test data
- In-memory HSQLDB implementation : search time 38ms
- In-memory CQEngine implementation : search time 3ms
-

Collection indexing

The next set of tests looked at different ways of indexing the data in the CQEngine Collections.

The tests looked at three different indexing schemes for the inner ConcurrentIndexedCollection of Positions. The first scheme simply created separate indexes on right ascension and declination.

```
positions.addIndex(
    NavigableIndex.onAttribute(
        ZoneMatcherImpl.POS_RA
        )
    );

positions.addIndex(
    NavigableIndex.onAttribute(
        ZoneMatcherImpl.POS_DEC
        )
    );
```

The second scheme created separate indexes, but used a quantized index on right ascension.

```
positions.addIndex(
    NavigableIndex.withQuantizerOnAttribute(
        DoubleQuantizer.withCompressionFactor(
            5
            ),
        ZoneMatcherImpl.POS_RA
        )
    );

positions.addIndex(
    NavigableIndex.onAttribute(
        ZoneMatcherImpl.POS_DEC
        )
    );
```

The third scheme created a combined CompoundIndex on right ascension and declination together.

```
positions.addIndex(
    CompoundIndex.onAttributes(
        ZoneMatcherImpl.POS_RA,
        ZoneMatcherImpl.POS_DEC
        )
    );
```

These tests showed a significant advantage for the separate simple indexes for this particular use case:

- 12,587,009 rows of test data
- Combined indexes : average 42ms
- Quantized ra index : average 21ms
- Separate indexes : average 0.45ms

The results of the indexing tests match the way that the algorithm was implemented in the database version and the native Java version.

The HSQLDB database implementation used a single SQL query to perform all three stages of the zone algorithm, selecting the zone, selecting positions within the zone based on ra and dec and then performing the final distance calculation all in one database query. It therefore makes sense that the combined and complex indexes performed better than the separate single value indexes for this case.

The CQEngine Collections implementation performed the three stages of the zone algorithm, selecting the zone, selecting positions within the zone and then performing the final distance calculation as separate steps in the Java program. It therefore makes sense that using three separate indexes gave the best performance for this implementation.

Zone height

The final set of tests looked at the relationship between the size of the zones, search radius and performance. Due to limited time we were unable to develop a detailed model of the relationship . However we were able to confirm the general rule described in Gray et al. That the algorithm produced the best results when the zone height was close to or equal to the search radius.

Data size	Zone height	Search radius	Search time (ms)
12,587,009	0.25	0.015625	1.665
12,587,009	0.125	0.015625	2.312
12,587,009	0.0625	0.015625	0.669
12,587,009	0.03125	0.015625	0.528
12,587,009	0.015625	0.015625	0.419
12,587,009	0.0078125	0.015625	0.450
12,587,009	0.00390625	0.015625	0.607

Although some test results suggested that this may not always be the case and the true relationship may depend on additional factors such as the size or structure of the catalog being compared.

Summary of results

Based on the test results, this set of experiments have been able to demonstrate a cross match algorithm that is capable of meeting the target data rate of 1,000 alerts per second for catalog sizes of the order of 60 million sources. Further testing will be needed to demonstrate how this capability can be scaled to handle larger data sets.

Further experiments

There is more that could be done to develop these experiments further, extending the size of the test data set, optimizing the indexing and exploring the relationship between zone size, search radius and search performance.

There is also more work to do to explore ways to make the system scalable and fault tolerance.

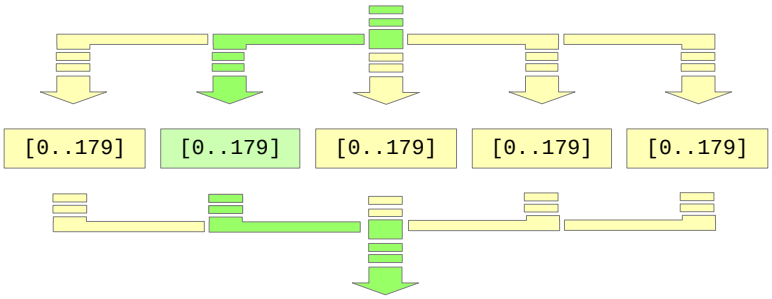
To meet the scalability and fault tolerance criteria we need to look at ways to distribute the crossmatch searches over multiple machines.

There are two aspects to the scalability question:

- How to increase the processing rate.
- How to increase the size of the catalog.

Multiple copies

One way of distributing the data over multiple nodes would be for every node to have a full copy of the catalog. In this scenario, given a data range of 0..180, then all of the nodes would have the full data range in memory, and each input alert would only need to be sent to one of the nodes in the cluster.



Processing cross matches for multiple alerts could be done in parallel, scaling the data rate in direct relation to the number of nodes.

The Kafka ‘at least once’ delivery mode supports this pattern, using commit messages to confirm that each input alert is delivered to one processing node. This configuration would also meet the fault tolerance criteria. As each node is identical, the system could operate with spare capacity ready to take the load if any of the nodes fails.

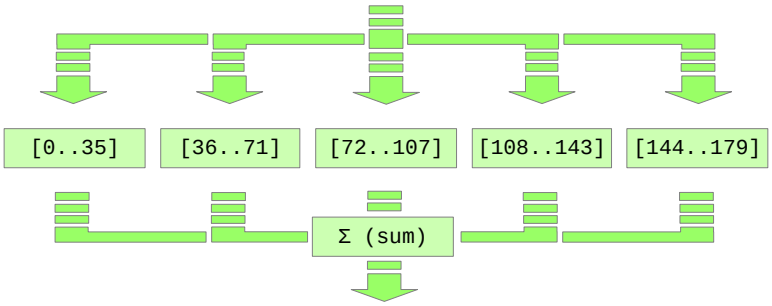
In theory if each node is capable of processing <*n*> alerts per second, then a set of four nodes sharing the data between them would be capable of handling *4n* alerts per second. Further testing will be needed to determine what the scaling is in reality, and how much time and resources are lost due to overheads.

The disadvantage of this configuration is that every node in the system has to have a complete copy of the catalog. Limiting the size of dataset to the available memory in a single node.

Multiple slices

Another way of distributing the data would be to assign a subset of the data to each node.

Processing an all sky cross match would follow the map-reduce pattern. All of the alerts would be sent to all of the nodes, each node would perform the cross match against its local subset of the data and then the results from all the nodes would be aggregated to generate the final result.



The problem with this configuration is that passing all of the partial results to a single aggregation node means the aggregation node itself becomes a bottleneck in the system.

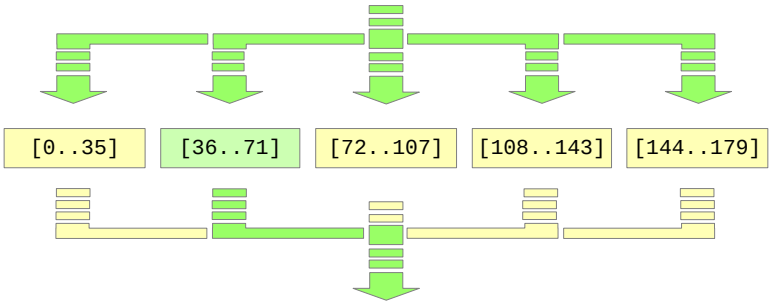
However, it may be possible to configure the system so that we do not need the aggregation node.

If we add a filtering mechanism to the nodes so that they do not respond to alerts outside the range for their slice identifier, then only the node(s) assigned to that slice would produce a result for an alert that falls within that slice. In which case, if there is only one result from one node per alert there is no need to aggregate the results from any of the other nodes.

For example, if we divide a position range from 0 to 180 into 5 slices

```
slice [0] = [0..35]
slice [1] = [36..71]
slice [2] = [72..107]
slice [3] = [108..143]
slice [4] = [144..179]
```

If we assign each slice to a node based on these values, then an alert at 24 would be assigned to the node for slice [1] and an alert at 102 would be assigned to the node for slice [2].



None of the other nodes would respond to these alerts, and so there would only be one result for each alert. In which case there is no need for the aggregating reduce step of the map-reduce pattern.

In order to cover the edge case of alerts close to the boundary between the slices, the actual data in each slice would need to be extended to include a search radius either side of the range for that slice. Extending our example to include a search radius of 2, then the ranges for the slice identifiers and the actual data within each slice would be:

```
slice [0] = [0..35] contains [-2..37]
slice [1] = [36-71] contains [34..73]
slice [2] = [72-107] contains [70-109]
slice [3] = [108-143] contains [106-145]
slice [4] = [144-179] contains [142-181]
```

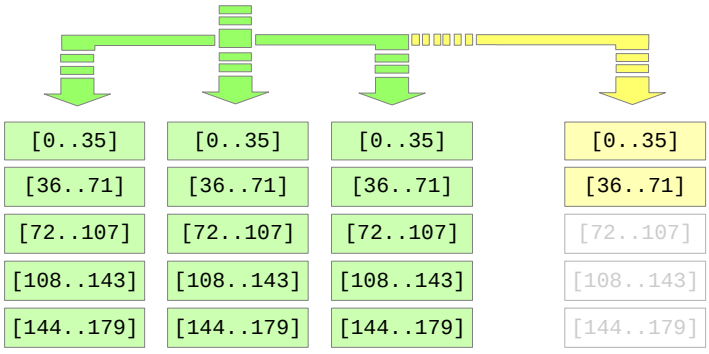
This configuration would enable the system to scale to handle increases in the size of the catalog. However, it would be up to us to provide the functionality required to distribute the data between the nodes and to re-balance the data when a node is added or removed from the system.

Although the concepts for this configuration are fairly well described, further experiments will be needed to develop and test a data distribution system before we could complete the evaluation of this design.

Nested configurations

The multiple copies configuration addresses the scalable data rate and fault tolerance criteria, and the multiple slices configuration addresses the scalable data size criteria.

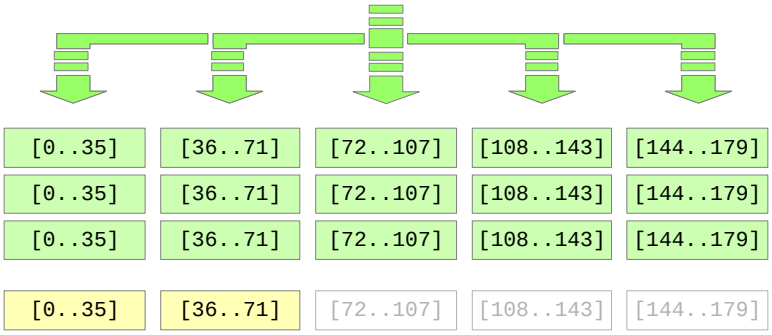
Combining the two configurations would address both sets of criteria. However there are trade offs to be made in terms of complexity, reliability, and the size of the replicated units.



For example, if the system is configured as three copies each containing an array of five slices, then scaling the system to increase the data processing rate would require adding a full copy containing a complete set of five slices.

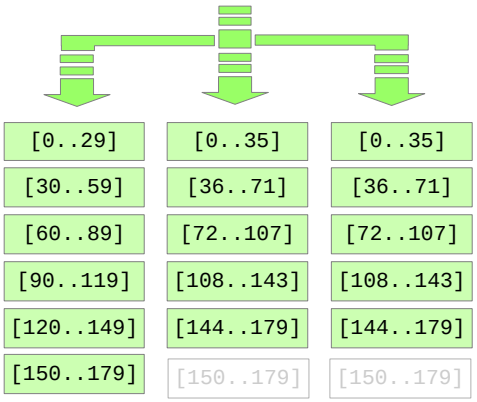
The fourth copy would have to be fully populated with nodes for all of the slices before it could be enabled. If a copy containing a partial set of slices is enabled then data routed to that copy destined for the missing slices would be lost.

Alternatively, if the system is configured as five slices each containing three copies, then the scaling the system to increase data processing rate can be done more gradually.



Adding a fourth copy to two of the slices would increase the data rate for those particular slices, but would not cause side effects for the other slices. Adding additional copies to each of the slices in turn would gradually increase the system performance.

However, to scale the system storage capacity the reverse would be true.

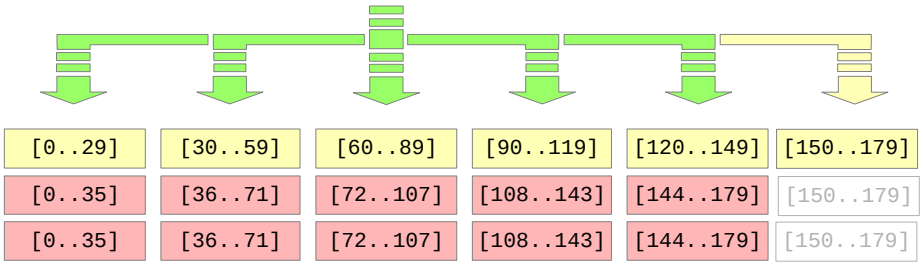


In the first example, configured as three copies of five slices, the system can cope with different numbers of slices within the copies, so slices can be added to each of the copies one at a time.

The process of re-balancing the data within a copy may cause some disruption, but it would not require the whole system to be

interrupted.

In the second example, configured as five slices containing three copies, all of the copies within a slice have to be the same size.



So it would not be possible to expand the number of slices from five to six without updating all of the copies at the same time.

Although the description of the nested configurations sounds complex, in practice all of the routing logic to pass the alerts to all of the slices, or only one of the copies are already implemented in Kafka. Implementing the nested configurations could be done using the standard Kafka consumer and producer components to route the alerts to either all or one of the nodes depending on the configuration.

Future tests are planned which will develop prototypes for each of these solutions to evaluate whether they provide the expected gains in processing bandwidth, storage capacity and fault tolerance, and what the additional costs involved in managing the overall system are.

Data caching

Another way to solve the data size problem is to to treat the in-memory data as a cache for the full catalog and only load a subset into memory at any one time. This requires a mechanism for dynamically updating the contents of the cache to match the incoming data.

On demand loading

One way of implementing this is to load the data into memory on demand. Each time the Java code tries to load a block of data it checks to see if the data is in memory yet, and if not it loads the data from disc before continuing. The side effect of this is that every time there is a cache-miss the system would have to pause while the required data is loaded. This would in turn have an impact on system performance, making it dependent on I/O data rate.

Time based loading

An alternative approach would be to load cache based on what we know about the telescope, its physical location and its field of view. The region of sky a ground based instrument can see changes over the course of a night as the Earth rotates. Based on this we can calculate the region of the sky visible to the telescope at any given point in time, and hence the subset of data in the catalog that could potentially match the alerts in the live stream.

Note that this is not based on where the telescope is pointing, but simply the region of sky it is physically possible to see from that location on Earth at that time.

A timer based background process could calculate the visible region and update the subset of the catalog in memory once an hour, loading new data before it becomes visible and removing old data after it drops out of view. As it is only run once an hour it is not time critical, so it could use a more

complex calculation and selection process to load the a data from a conventional database.

This solution avoids the delays to the alert processing caused by a cache miss in an on-demand system that loads blocks of data when they are needed, reducing the impact of I/O data rate on system performance.

Instrument pointing

– calculate the cached data based on knowledge of the instrument pointing --

Cache expiry

Several methods for selecting the subset of data to load into the cache. We also need to remove data at an equal rate or it will rapidly exceed the available space.

Loading selectivity

Depending on how accurate the data selection process is, there will be some degree of overlap between the existing data in the cache and the new set of data being loaded for the next hour.

For a general selection like the visible sky calculation described above, there will be a fairly large area of overlap between the visible sky for this hour and the visible sky for the next hour. Only a small crescent shape at the leading edge will be new, and an equivalent shape at the trailing edge will no longer be visible.

We could just load the whole of the visible area every hour and simply expire data based on timestamp. If we re-scan the visible area each hour and update the time stamp on objects already in the cache, then we can identify the objects that need to be removed from the cache simply by iterating the list of objects in the cache and releasing any objects whose expiry time has been reached.

However, this method for loading the cache relies on extremely fast loading and cross matching of the data. Assuming that majority of objects in a catalog will not be transient, then the cache ingest rate for a simple visibility calculation is likely to be be several orders of magnitude larger than the transient data rate.

At the other end of the scale is to calculate the delta changes in visibility, only load or drop data from the areas that change. Loading objects that are in the area that is not currently visible will become visible in the next hour, and only dropping objects that are in the area that was visible up to a hour ago but is is now no longer visible.

This is requires an much more accurate calculation of the visible areas, an accurate method of calculating the difference between two areas, an accurate method of selecting the data to load from the catalog that matches the new area, and an equivalent method for selecting the data to remove from the cache.

Further experiments

Further experiments are planned which will develop prototypes for each of these solutions to see where the problems are and what performance gains they produce.

Kafka data store

An early experiments looked at the feasibility of using a Kafka service as a permanent data store.

Kafka is primarily designed to store a rolling buffer of data. The actual amount of data stored is set by a number of configuration parameters, including the amount of space it is allowed to use and the maximum length of time to keep the data for. By setting these configuration parameters to infinite, Kafka will store all the data it receives.

It does work. The built-in replication means that Kafka will store multiple copies of the data replicated over the machines within a cluster making it a reliable and robust data store.

However, we found two issues which make Kafka a less than ideal data store. The first issue is with indexing. Kafka is primarily designed as a temporary store and forward message processing system, not as a long term data store. As a result support for indexing and selecting subsets of the data are limited.

The primary use case of Kafka indexing is to subscribe to a stream of data starting from a specific [start date] or [offset] in the stream. This will generate a ***live*** stream starting from the specified location in the topic history, and stream all the messages since that point, ***followed by any new messages for that topic***.

The important thing is that the streams will not stop when they reach the end of the current data - t***here is no end-of-stream marker in Kafka***. The streams will stay open, waiting to forward new messages being sent to that topic.

If the intention is to request a subset of messages between [start] and [end] offsets, then it is up to the client to realize that it has reached the last of the records it wanted to see and to close the connection from the client end. Unless told otherwise, Kafka will continue to wait and forward new incoming messages to the result stream.

There is work in progress in the data analytics community to combine live data streams and historical archives, but in all the cases we have found they use a separate database, e.g Cassandra or Druid, to store the historical data.

The second issue with using Kafka as a long term data store is a more practical issue concerning data validation.

When Kafka is started it checks its data store directories to find all the stored streams on disc. If the service was not shut down *just so*, the server will default to checking and re-indexing any stored data that it finds before registering the service as live and ready to accept new data.

This is a very conscientious and safe way of operating, intended to prevent services with corrupt data sharing that data with other nodes in the cluster and potentially corrupting their data too. This works well if Kafka is configured as a short term store and forward messaging system.

However, if Kafka is configured as a long term data store, the propensity to re-index all of its data means that the service will take longer and longer to start up as the size of the data grows. If the delay at start up becomes longer than the heart-beat interval used to maintain the register of active nodes in the cluster the a service will be dropped from the cluster as unresponsive before it manages to index all of its data. In the worst case scenario this can result in the other members of the cluster attempting to re-balance the remaining replicas of the data that is on the missing server to maintain the target replication factor. If/when the dropped service finally manages to join the cluster it may discover that all of the data it carefully re-indexed is no longer needed because the other services have re-balanced the replicas between themselves making it redundant.