Qserv: A distributed shared-nothing database for the LSST catalog

Daniel L. Wang[†]
danielw@slac.stanford.edu

Serge M. Monkewitz[*]
smm@ipac.caltech.edu

Kian-Tat Lim[†]
ktl@slac.stanford.edu

Jacek Becla[†]
becla@slac.stanford.edu

[†]SLAC National Accelerator Laboratory
2575 Sand Hill Road, MS 97
Menlo Park, CA 94303

[*]Infrared Processing and Analysis Center
California Institute of Technology
Mail Code 100-22
Pasadena, CA 91125

ABSTRACT

The LSST project will provide public access to a database catalog that, in its final year, is estimated to include 26 billion stars and galaxies in dozens of trillion detections in multiple petabytes. Because we are not aware of an existing open-source database implementation that has been demonstrated to efficiently satisfy astronomers' spatial self-joining and cross-matching queries at this scale, we have implemented Qserv, a distributed shared-nothing SQL database query system. To speed development, Qserv relies on two successful open-source software packages: the MySQL RDBMS and the Xrootd distributed file system. We describe Qserv's design, architecture, and ability to scale to LSST's data requirements. We illustrate its potential with test results on a 150-node cluster using 55 billion rows and 30 terabytes of simulated data. These results demonstrate the soundness of Qserv's approach and the scale it achieves on today's hardware.

Keywords

parallel, shared-nothing, database, distributed, MPP, file system

1. INTRODUCTION

The scientific community has often been at the forefront of computing technology, pushing limits in processing power, data volume, network bandwidth, and software. The Large Synoptic Survey Telescope (LSST)[9] will push the limits of technology in gathering wide, frequent, and detailed images of the sky and will result in the most comprehensive catalog of stars and galaxies ever created. The LSST camera will record new 3 gigapixel images every 15 seconds all night, every night (except in downtime) over 10 years of operation. Each year, the cumulative collected images are processed into data release catalogs of stars, galaxies, and other celestial bodies, which are made available to professional astronomers, academics, and the general public through a query access system. This paper describes our work in building a prototype for such a system.

2. PROBLEM

LSST will need to provide ad-hoc user query access to its data release catalogs. Table 1 describes the estimated sizes for key tables in LSST's last catalog.

Key catalog tables

	# rows	row size[a]	footprint[a]
Object	26×10^9	2kB	48TB
Source	1.8×10^{12}	650B	1.3PB
ForcedSource	21×10^{12}	30B	620TB

Table 1: Estimates for LSST's final data release

[a] Raw storage bytes needed (neglecting compression and database overheads e.g., indexing, paging, fragmentation, alignment, and metadata.)

2.1 Requirements

In general, the production data access system must be (a) incrementally scalable, (b) reliable and available, and (c) low-cost. The low cost requirement motivates a strong preference towards an open-source solution, since the solution must be maintained over a minimum of 10 years and be easily available for any astronomer to use.

In terms of usage, the query access system must support a continuous concurrent load of about 50 "low volume" queries, 20 "high volume" queries, and 1 "super high volume" query. The low volume class includes light interactive use, with response times less than 10 seconds. The high volume class includes moderately complex (e.g., statistical aggregation) queries over the full sky that should complete in 1 hour. Long analyses, including near-neighbor correlations, are classified as super high volume and should complete in less than 10 days. In other words, the system should support a mixed load that ranges from interactive simple queries to more batch-like trillion-row joins.

3. ALTERNATIVES

Several solutions were considered before we embarked on Qserv development.

Mainstream RDBMS. Using an off-the-shelf RDBMS would have conferred several advantages: mature code, broad functionality, a development ecosystem, a user community, support structures, and developer experience. However, none of

Copyright is held by the author/owner(s)
SC'11, November 12-18,2011,Seattle,Washington,USA.
ACM 978-1-4503-0771-0/11/11

the single-node solutions could provide enough performance at our expected scale.

Teradata/Greenplum. Although our requirements for long-term support and no-cost distribution make proprietary solutions unattractive, Teradata[15] and Greenplum[5] offer distributed shared-nothing database solutions that may scale to LSST's data sizes, though neither has spherical spatial join support to our knowledge.

Hadoop/HDFS. We considered storing the catalog in HDFS[12] and using Hadoop's map-reduce model for queries, but discarded the idea for a number of reasons. The catalog is well defined in a relational model and it is unclear how to efficiently represent it in HDFS without discarding its structure. Without an efficient representation, each query would require a full read of the catalog, which is too expensive when a database with indexing could return an answer more cheaply for a large class of queries. Another disadvantage is Hadoop's current imposed latency—its minimum job overhead would be prohibitive for otherwise cheap queries. Finally, astronomers were (somewhat) reluctantly trained on SQL with SDSS[10] and would be unlikely to learn yet another programming model (map-reduce).

HBase/Hive. We considered using HBase[7] and Hive[8] since they aim to provide fast query access for relational data. We did not test HBase since it did not support relational joins, although its fast row look-up might still be leveraged in other ways. We tested Hive on a number of queries, but found that while it did scale somewhat for simple queries, its lack of indexing and spatial join support were significant hindrances. The lack of indexing meant that selections on tables were executed as full table scans. Hive's basic support for joins meant that while it could parallelize the reading of rows, the join aggregation phase was not parallelized and was not sped up with additional processors.

Other "NoSQL". Our requirements seem like they could be satisfied with a NoSQL[2, 11] approach—scalability, performance, latency, and fault-tolerance over transactions and strict consistency. However, other than the Hadoop-based solutions, none are sufficient as integrated solutions for scalable catalog query access (though they may be useful as components in a larger solution), and none have indexing and efficient spatial join support. None have a query-able interface like SQL that could be used to push analytics to data for parallel execution—they would thus require a parallel computation framework that may not be able to leverage and optimize for data locality and bandwidth.

4. DESIGN CONCEPTS

4.1 Distributed and parallel

Qserv was implemented with a model of distributing and parallelizing computation among largely autonomous worker nodes. In this model, many (even all) workers may contribute to the result of a single user task, while having no direct knowledge of each other and completing their assigned work without data or management from their peers.

4.2 Shared-nothing

LSST's catalog will involve dozens of petabytes[1] spread over many nodes. Operating under the assumption that local-attached storage always has the highest bandwidth per unit cost, a shared-nothing architecture allows each node to focus on its own work on its own data. There is no contention at a shared storage apparatus for bandwidth or IOPS, and there is no wasted time spent coordinating other than receiving work and returning results. Qserv's interconnection fabric can be constructed of simple low-cost commodity networking hardware rather than specialized high-bandwidth or low-latency hardware.

4.3 Shared scanning

In estimating costs and designing Qserv, it became clear that I/O bandwidth from disks would be the greatest bottleneck. As such, Qserv forgoes heavy indexing in most cases because index use often yields random row-reading from disk. When tables are so large that no significant fraction can fit in memory, it is cheaper to read sequentially from disk than to seek for particular rows (especially when the index itself is out-of-memory). Qserv limits its use of indexing to particular use cases where indexing can provide substantial benefit: spatial indexing for sharding and spatial queries and objectId indexing to satisfy ad-hoc queries for particular objects.

Now with table-scanning being the norm rather than the exception and each scan taking a significant amount of time, multiple full-scan queries would randomize disk access if they each employed their own full-scanning read from disk. Shared scanning (also called convoy scheduling)[6] shares the I/O from each scan with multiple queries. The table is read in pieces, and all concerning queries operate on that piece while it is in memory. In this way, results from many full-scan queries can be returned in little more than the time for a single full-scan query.

4.4 Partitioning

Data must be partitioned among nodes in a shared-nothing architecture. While some *sharding* approaches partition data based on a hash of the primary key, this approach is unusable for LSST data since it eliminates optimizations based on celestial objects' spatial nature.

Sharded data and sharded queries. Qserv divides data into spatial partitions of roughly the same area. Since objects occur at a similar density (within an order of magnitude) throughout the celestial sphere, equal-area partitions should evenly spread a load that is uniformly distributed over the sky. If partitions are small with respect to higher-density areas and spread over computational resources in a non-area-based scheme, density-differential-induced skew will be spread among multiple nodes.

With data in separate physical partitions, user queries are

[1]Table 1 highlights one table in one data release and does not include database overheads (e.g., paging, indexing) or replication. Each data release will consist of many tables and 11 data releases are planned.

themselves fragmented into separate physical queries to be executed on partitions. Each physical query's result can be combined into a single final result.

Two-level partitions. Determining the size and number of data partitions may not be obvious. Queries are fragmented according to partitions so an increasing number of partitions increases the number of physical queries to be dispatched, managed, and aggregated. Thus a greater number of partitions increases the potential for parallelism but also increases the overhead. For a data-intensive and bandwidth-limited query, a parallelization width close to the number of disk spindles should minimize seeks while maximizing bandwidth and performance.

From a management perspective, more partitions faciliate re-balancing data among nodes when nodes are added or removed. If the number of partitions were equal to the number of nodes, then the addition of a new node would require the data to be re-partitioned. On the other hand, if there were many more partitions than nodes, then a set of partitions could be assigned to the new node without re-computing partition boundaries.

Smaller and more numerous partitions benefit spatial joins. In an astronomical context, we are interested in objects near other objects, and thus a full $O(n^2)$ join is not required—a localized spatial join is more appropriate. With spatial data split into smaller partitions, a SQL engine computing the join need not even consider (and reject) all possible pairs of objects, merely all the pairs within a region. Thus a task that is naively $O(n^2)$ becomes $O(kn)$ where k is the number of objects in a partition.

In consideration of these trade-offs, two-level partitioning seems to be a conceptually simple way to blend the advantages of both extremes. Queries can be fragmented in terms of coarse partitions ("chunks"), and spatial near-neighbor joins can be executed over more fine partitions ("subchunks") within each partition. To avoid the overhead of the subchunks for non-join queries, the system can store chunks and generate subchunks on-demand for spatial join queries. On-the-fly generation for joins is cost-effective due to the drastic reduction of pairs, which is true as long as there are many subchunks for each chunk.

Overlap. As discussed above, dividing objects spatially eliminates joining pairs of objects that are distant. However, a strict partitioning also eliminates nearby pairs where objects from adjacent partitions are paired. To produce correct results under strict partitioning, nodes need access to objects from outside partitions, which means that data exchange is required. To avoid this, each partition can be stored with a precomputed amount of overlapping data. This overlapping data does not strictly belong to the partition but is within a preset spatial distance from the partition's borders. Using this data, spatial joins can be computed correctly within the preset distance without needing data from other partitions that may be on other nodes.

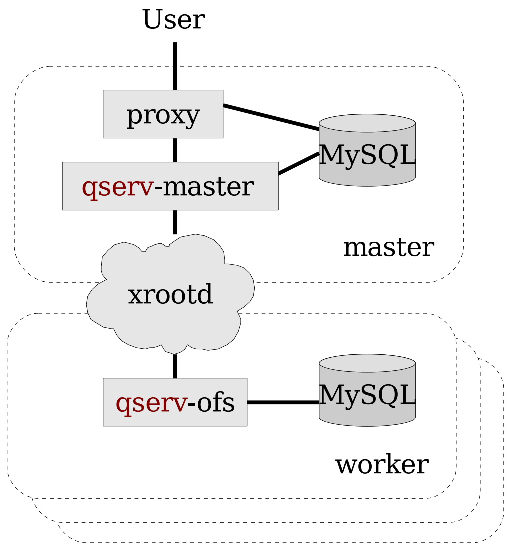

Figure 1: Component connections in Qserv

Spherical geometry. Support for spherical geometry is not common among databases and spherical geometry-based partitioning was non-existent in other solutions when we decided to develop Qserv. Since spherical geometry is the norm in recording positions of celestial objects (right-ascension and declination), any spatial partitioning scheme for astronomical objects must account for its complexities.

5. IMPLEMENTATION

We implemented Qserv as a shared-nothing distributed database system. Qserv is optimized for scalability in data size and read access. Support for updates has not been implemented. Shared scanning is planned for implementation later this year.

5.1 Components

5.1.1 MySQL

To control the scope of effort, Qserv uses an existing SQL engine, MySQL, to perform as much query processing as possible. MySQL is a good choice because of its active development community, mature implementation, wide client software support, simple installation, lightweight execution, and low data overhead. MySQL's large development and user community means that expertise is relatively common, which could be important during Qserv's development or long-term maintenance in the years ahead. MySQL's MyISAM storage engine is also lightweight and well-understood, giving predictable I/O access patterns without an advanced storage layout that may demand more capacity, bandwidth, and IOPS from a tightly constrained hardware budget.

It is worth noting, however, that Qserv's design and implementation do not depend on specifics of MySQL beyond glue code facilitating results transmission. Loose coupling is maintained in order to allow the system to leverage a more advanced or more suitable database engine in the future.

5.1.2 Xrootd

The Scalla/Xrootd distributed file system[4] is used to provide a distributed, data-addressed, replicated, fault-tolerant communication facility to Qserv. Re-implementing these features would have been non-trivial, so we wanted to leverage an existing system if possible. Xrootd has provided scalability, fault-tolerance, performance, and efficiency for many years in serving large files in the high-energy physics community and elsewhere, and its relatively flexible API enabled its use as a more general communication medium instead of a file system. Since it was designed to serve large data sets, we were confident that it could mediate not only query dispatch communication but bulk transfer of results.

A Scalla/Xrootd cluster is implemented as a set of data servers and one or more redirectors. A client connects to a redirector, which acts as a caching namespace look-up service that redirects clients to appropriate data servers. In Qserv, Xrootd data servers become Qserv workers by plugging custom code into Xrootd as a custom file system ("ofs plugin") implementation. The Qserv master dispatches work to workers by writing to partition-addressed Xrootd paths and reads results from hash-addressed Xrootd paths.

5.2 Partitioning

In Qserv, large spatial tables are fragmented into spatial pieces in the two-level partitioning scheme discussed in section 4.4. The partitioning space is a spherical space defined by two angles ϕ and θ (in astronomy, right ascension/α and declination/δ). For example, the **Object** table is fragmented spatially, using the right-ascension and declination coordinates. On worker nodes, these fragments are represented as tables named **Object_CC** and **Object_CC_SS** where CC is the "chunk id" (first-level fragment) and SS is the "sub-chunk id" (second-level fragment of the first larger fragment. Subchunk tables are built on-the-fly to optimize performance of spatial join queries. Large tables are partitioned on the same spatial boundaries where possible to enable joining between them.

5.3 Query generation

Partitioning is hidden from the user, so Qserv rewrites user queries for execution on chunk and sub-chunk tables on worker nodes. We have extended Lubos Vnuk's Sql2SQL[17] grammar to handle the necessary query token and phrase detection to extract characteristics necessary for generating "chunk queries" for dispatch.

In Qserv, query parsing serves several functions:

- Detect spatial restrictions. Queries that include spatial restriction do not need to be dispatched on all chunks. This prevents spatial queries from becoming full-sky queries and saves significant worker load as well as overhead for dispatch and management.

- Detect index opportunities. While only one column is indexed in our case, indexing is crucial for optimizing an important class of queries. See section 5.5.

- Detect database and table references. Each reference is detected and instrumented so that it may be rewritten. Not all tables are partitioned, and database references are sometimes rewritten as well. Detection also facilitates access restriction.

- Detect aliases and joins. SQL aliases are common, especially in join syntaxes and must be appropriately managed during rewriting.

- Other preparation for results merging and aggregation.

Example. Consider a user query:

```
SELECT AVG(uFlux_SG)
  FROM Object
  WHERE qserv_areaspec_box(0.0, 0.0, 10.0, 10.0)
    AND uRadius_PS > 0.04;
```

The AVG(uFlux_SG) function call is converted into a SUM(uFlux_SG) and COUNT(uFlux_SG) pair for chunk queries and SUM('SUM(uFlux_SG)') / SUM('COUNT(uFlux_SG)') to aggregate the resulting rows after results from all chunks have been gathered.

The reference to the Object table is converted to LSST.Object_CC, where CC is substituted appropriately for each chunk. The "LSST." database qualifier is added from the user database context and is necessary for the query to operate in the different context available on worker nodes.

The `qserv_areaspec_box(0.0, 0.0, 10.0, 10.0)` pseudo-function call is used to select a set of chunks over $0.0 < \phi < 10.0$ and $0.0 < \theta < 10.0$, and is rewritten to operate using a user-defined function installed on worker database instances. The Object table is partitioned where (ϕ, θ) are (ra_PS and decl_PS) so the call is rewritten as `qserv_ptInSphericalBox(ra_PS, decl_PS, 0.0, 0.0, 10.0, 10.0) = 1`.

Qserv does not currently support SQL subqueries.

5.4 Dispatch

A MySQL Proxy[16] wraps the qserv frontend so that queries can be submitted using any MySQL-compatible client or library. The frontend's generated queries are dispatched using two file-level transactions on Qserv's Xrootd cluster. The first transaction consists of opening a particular path for writing, writing the chunk query, and closing the file. The path contains a specified chunkId and has the format: **xrootd://<*manager_ip:port*>/query2/**CC, where CC is the chunkId. The second transaction reads query results and consists of opening a path for reading, reading until EOF, and closing the file. The second path specifies the hash of the chunk query written in the original chunk query and has the format: **xrootd://<*worker_ip:port*>/result/**H, where H is the MD5 hash, represented via 32 hexadecimal digits in ASCII.

Chunk Query Representation. The format of a chunk query is given as a set of SQL query statements where the first line is a comment and indicates sub-chunk dependency.

```
-- SUBCHUNKS: <subChunkId0>[, <subChunkId1>[, ..]]
<SQL statement 1>;
[<SQL statement 2>;]
...
```

The SUBCHUNKS line indicates the list of required subchunks for the query. The worker must generate the appropriate subchunk tables prior to executing the SQL statements, but is free to drop the tables afterwards. This enables the worker to cache subchunk tables, although the current implementation does not cache them.

Query Results Transfer. Results from a chunk query are transferred as SQL statements. The worker executes `mysqldump` on the result table and the resulting byte stream is read byte-for-byte by the master, which executes the SQL statements to load results into its local database. After each result table is loaded, it is merged into a table which serves as the final result table for non-aggregating queries. When aggregation is needed, an aggregation query is executed on this table to produce the final result table.

Using `mysqldump` introduces overheads, but is the only user-level method provided by MySQL to transfer tables between database servers. We are considering implementing a more efficient method as development resources permit.

5.5 Indexing

By construction, Qserv's implementation of two-level spatial partitioning provides coarse spherical indexing so that spatially-restricted queries can execute involving only the relevant spatial fragments. However, access that is not spatially restricted involves the entire table by default. Qserv also implements indexing for one particular column, objectId. This is implemented by including a three-column table in the frontend's metadata database that maps objectId to chunkId and subChunkId. When a query predicated on objectId (the indexed column) is submitted, the frontend executes queries on this table to compute the containing set of chunks. Chunk tables on workers' MySQL instances are also indexed by objectId so that indexed execution can be used on this containing set.

6. RESULTS

During its development, Qserv was tested in configurations of 1, 3, 15, 40, 100, and 150 nodes and tables of millions to billions of rows requiring up 30 TB of storage. In this paper, we will focus on a recent test using 150 nodes. Although we had access to the cluster for only 15 days and spent most of the time on data synthesizing, debugging, and development, we have collected results on several queries in 40, 100, and 150 node configurations.

6.1 Configuration

6.1.1 Hardware

We configured a cluster of 150 nodes interconnected via gigabit Ethernet. Each node had 2 quad-core Intel Xeon X5355 processors with 16GB memory and one 500GB 7200RPM SATA disk. Tests were conducted with Qserv SVN r21589, MySQL 5.1.45 and Xrootd 3.0.2 with qserv patches.

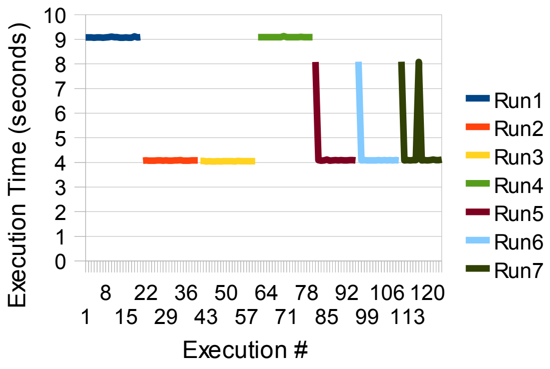

Figure 2: Performance on Low Volume 1

6.1.2 Data

We tested using a dataset synthesized by spatially replicating the dataset from a recent LSST data challenge ("PT1.1"). We used two tables: Object and Source.[2] These two tables are among the largest expected in LSST. Of these two, the Object table is expected to be the most frequently used. The Source table will have 50-200X the rows of the Object table, and its use is primarily confined to time series analyses that generally involve joins with the Object table.

The PT1.1 dataset covers a spherical patch with right-ascension between 358° and 5° and declination between -7° and 7°. This patch was treated as a spherical rectangle and replicated over the sky by transforming duplicate rows' RA and declination columns, taking care to maintain spatial distance and density by a non-linear transformation of right-ascension as a function of declination. This resulted in an Object table of 1.7 billion rows (2TB) and a Source table of 55 billion rows (30 TB)[3]. The Source table included only data between -54° and +54° in declination. The polar portions were clipped due to limited disk space on the test cluster. Partitioning was set for 85 stripes each with 12 sub-stripes giving a ϕ height of $\approx 2.11°$ for stripes and 0.176° for sub-stripes. Each chunk thus spanned an area of $\approx 4.5deg^2$, and each subchunk, $0.031deg^2$ This yielded 8983 chunks. Overlap was set to 0.01667° (1 arc-minute).

6.2 Queries

The current Qserv development focus is on features for scalability. We have chosen a set of test queries that demonstrate performance for both cheap queries (interactive latency), and expensive queries (hour, day latency). Runs of low volume queries ranged from 15 to 20 queries, while runs of high volume queries and super high volume queries consisted of only a few or even one query due to their expense. All reported query times are according to the command-line MySQL client (`mysql`).

- **Low Volume 1—Object retrieval**
  `SELECT * FROM Object WHERE objectId = <objId>`

[2]The schema may be browsed online at `http://lsst1.ncsa.uiuc.edu/schema/index.php?sVer=PT1_1/`
[3]Source for the duplicator is available at `http://dev.lsstcorp.org/trac/browser/DMS/qserv/master/trunk/examples`
.

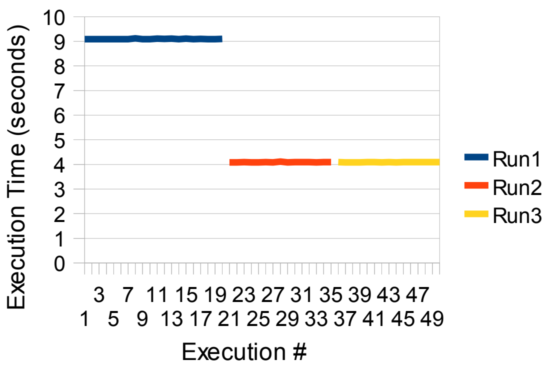

Figure 3: Performance on Low Volume 2

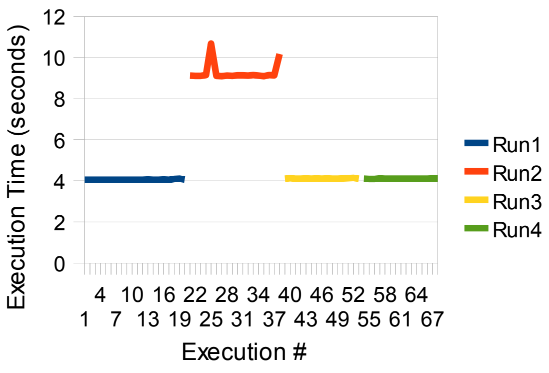

Figure 4: Performance on Low Volume 3

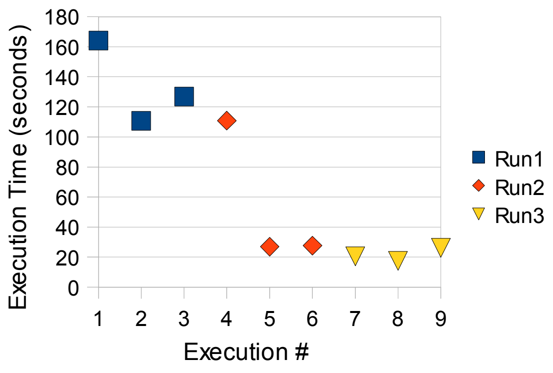

Figure 5: Performance of High Volume 1 (count)

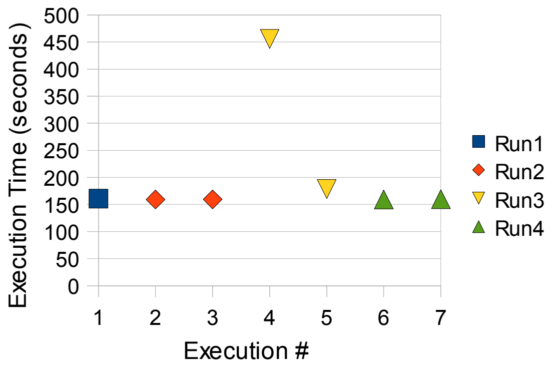

Figure 6: Performance of High Volume 2 (filter)

This query retrieves all information for a particular astronomical object. Queries of this type are expected to be very common. In testing, the objectId was randomized uniformly over the objects in the data set.

In Figure 2 we can see that performance of this query is roughly constant, taking about 4 seconds. Each run consisted of 20 queries. The slower performance of Runs 1 and 4, where each execution took 9 seconds, were the result of competing tasks in the cluster. We attribute the initial 8 second execution time in Run 5 and beyond to cold cache conditions (likely the objectId index) in the cluster.

- **Low Volume 2—Time series**

```
SELECT taiMidPoint, fluxToAbMag(psfFlux),
fluxToAbMag(psfFluxErr), ra, decl
FROM Source
WHERE objectId = <objId>
```

This query retrieves information from all detections of a particular astronomical object, effectively providing a time-series of measurements on a desired object. For testing, the objectId was randomized as for the Low Volume 1 query, which meant that null results were retrieved where the Source data was missing due to available space on the test cluster.

In Figure 3 we see that performance is roughly constant at about 4 seconds per query. Run 1 was done after Low Volume 1's Run 1 and we discount its 9 second execution times similarly as anomalous.

- **Low Volume 3—Spatially-restricted filter**

```
SELECT COUNT(*) FROM Object
WHERE ra_PS BETWEEN 1 AND 2
AND decl_PS BETWEEN 3 AND 4
AND fluxToAbMag(zFlux_PS) BETWEEN 21 AND 21.5
AND fluxToAbMag(gFlux_PS)-fluxToAbMag(rFlux_PS)
BETWEEN 0.3 AND 0.4
AND fluxToAbMag(iFlux_PS)-fluxToAbMag(zFlux_PS)
BETWEEN 0.1 AND 0.12;
```

This query asks how many objects of a certain color exist within a square degree box in the sky. The spatial location was randomized uniformly within ±20 degrees declination around the celestial equator. Limiting geospatial coverage is intended to limit performance variation due to varying object density that is a by-product of the spatial coverage of the original data set coupled with the simple data duplication technique we implemented. This query also exercises Qserv's rewriting of queries for simple aggregation.

In Figure 4 we see the same 4 second performance that was seen for the other low volume queries. Again, the ≈9 second performance in Run 2 could not be reproduced so we discount it as resulting from competing processes on the cluster.

- **High Volume 1—Count**

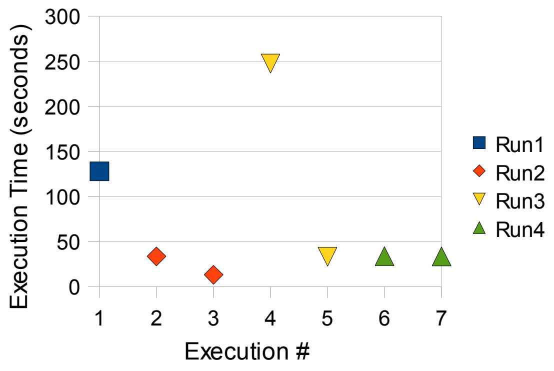

Figure 7: Performance of High Volume 3 (avg by chunk)

```
SELECT COUNT(*) FROM Object
```
This simple query exercises Qserv's query execution engine and illustrates the built-in cost of querying over all partitions in the sky. In theory, execution could exploit Qserv's objectId index in order to produce an object count, but the current implementation does not rely on any centralized index.

This COUNT(*) query was measured between 20-30 seconds, as shown in Figure 5. The slower performance during Run 1 can be attributed to interference of other processes (queries, maintenance) in the cluster.

- **High Volume 2—Full-sky filter**
```
SELECT objectId, ra_PS, decl_PS,
uFlux_PS, gFlux_PS, rFlux_PS, iFlux_PS,
zFlux_PS, yFlux_PS
FROM Object
WHERE fluxToAbMag(iFlux_PS) -
fluxToAbMag(zFlux_PS) > 4
```
This query retrieves all objects of a certain color beyond a threshold over the entire sky. It is a full table scan query over the Object table, and is an example of a simple query that would be batched into a shared-scan because of its I/O intensity. Figure 6 illustrates its stable performance over 150 nodes: 2.5 to 3 minutes per query. This may not be a fair measure of performance, since we have not controlled for caching behavior in MySQL and the operating system. The 7 minute time in Run 3 may be a more accurate measure of uncached execution time, and the shorter time a measure of overhead in a cached collection of the ≈70k rows of results.

Using the on-disk data footprint (MySQL's MyISAM .MYD, without indexes or metadata) of the Object table (1.824×10^{12} bytes), we can compute the aggregate effective table scanning bandwidth. Run 3's 7 minute execution yields 4.0GB/s in aggregate, or 27MB/s per node, while the other runs yield approximately 11GB/s in aggregate, or 76MB/s per node. Since each node was configured to execute up to 4 queries in parallel, Run 3's bandwidth is more realistic, given seek activity from competing queries and the disk manufacturer's reported theoretical transfer rate of 98MB/s[14].

- **High Volume 3—Density**
```
SELECT count(*) AS n, AVG(ra_PS), AVG(decl_PS),
chunkId
FROM Object
GROUP BY chunkId
```
This query computes statistics for table fragments (which are roughly equal in spatial area), giving a rough estimate of object density over the sky. It illustrates more complex aggregation query support in Qserv. This query is of similar complexity to High Volume 2, but Figure 7 illustrates measured times significantly faster, which is probably due to reduced results transmission time. As mentioned for HV2, cache behavior was not controlled, but the 4 minute time in Run 3 may be close.

- **Super High Volume 1—Near neighbor**
```
SELECT count(*)
FROM Object o1, Object o2
WHERE qserv_areaspec_box(-5,-5,5,-5)
AND qserv_angSep(o1.ra_PS, o1.decl_PS,
o2.ra_PS, o2.decl_PS) < 0.1;
```
This query finds pairs of objects within a specified spherical distance which lie within a particular part of the sky. Over two randomly selected 100 square degree areas, the execution times were about 10 minutes (667.19 seconds and 660.25 seconds). The resultant row counts ranged between 3 to 5 billion. Since execution uses on-the-fly generated tables (see 5.2), the tables do not fit in memory, and Qserv does not yet implement caching, we expect caching effects to be negligible.

- **Super High Volume 2—Sources not near Objects**
```
SELECT o.objectId, s.sourceId, s.ra, s.decl,
o.ra_PS, o.decl_PS
FROM Object o, Source s
WHERE qserv_areaspec_box(224.1, -7.5, 237.1, 5.5)
AND o.objectId = s.objectId
AND qserv_angSep(s.ra, s.decl, o.ra_PS,
o.decl_PS) > 0.0045
```
This is an expensive query—an $O(kn)$ join over 150 square degrees between a 2TB table and a 30TB table. Each objectId is unique in Object, but is shared by 41 rows (on average) in Source, so $k \approx 41$. We recorded times of a few hours (5:20:38.00, 2:06:56.33, and 2:41:03.45). The variance is presumed to be caused by varying spatial object density over the three random areas selected.

6.3 Scaling

We tested Qserv's scalability by measuring its performance while varying the number of nodes in the cluster. To simulate different cluster sizes, the frontend was configured to only dispatch queries for partitions belonging to the desired set of cluster nodes. This varies the overall data size proportionally without changing the data size per node (200-300GB). We measured performance at 40, 100, and 150 nodes to demonstrate weak scaling.

6.3.1 Scaling with small queries

From Figures 8, 9, and 10, we see that execution time is unaffected by node count given that the data per node is

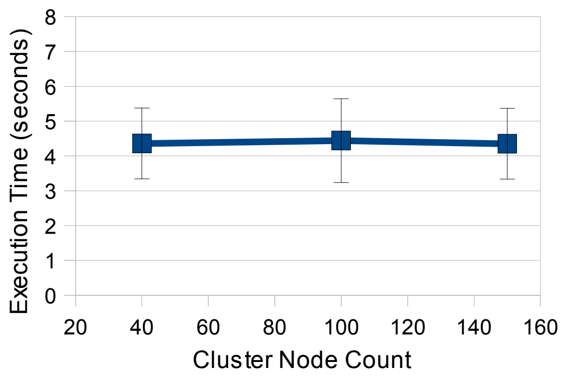

Figure 8: Low volume 1 query mean execution time vs node count (constant size per node)

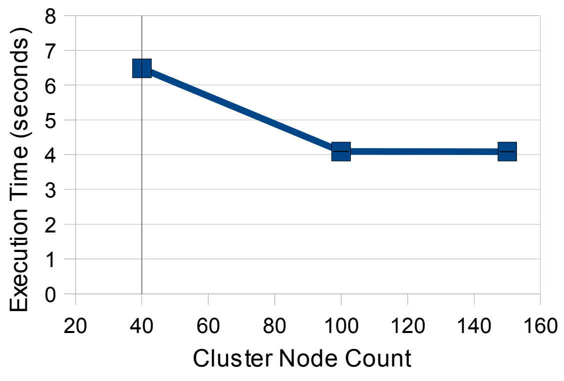

Figure 9: Low volume 2 query mean execution time vs node count (constant size per node)

constant. The spike in the 40-node configuration in Figure 9 is caused by 2 slow executions (23s and 57s); the other 28 executions in this node configuration had times ranging from 4.09 to 4.11 seconds. Since the slow executions represent less than 10% and the remaining >90% executions were so tightly bounded, the two slow executions are considered anomalous and likely due to unrelated but competing processes in the cluster.

The spike in mean execution time in Figure 10 is due to slower times for 6 of 24 executions at the 100-node configuration. 18 of the 24 executed in a range of 4.09 to 4.13 seconds. The slower 6 had times of 6.94, 8.10, 5.67, 7.25, 5.34, and 5.79 seconds. Excluding those 6 (25% overall) would have flattened the curve, but their presence demands further study. They are likely due to a combination of unrelated competing cluster activity and bugs in our implementation (3 of the 6 times occurred in series, indicating a longer-lasting transient).

6.3.2 *Scaling with expensive queries*

High Volume. If Qserv scaled perfectly linearly, the execution time should be constant when the data per node is constant. In Figure 11 the times for high volume queries show a slight increase. HV1 is a primarily a test of dispatch and result collection overhead and its time increases linearly

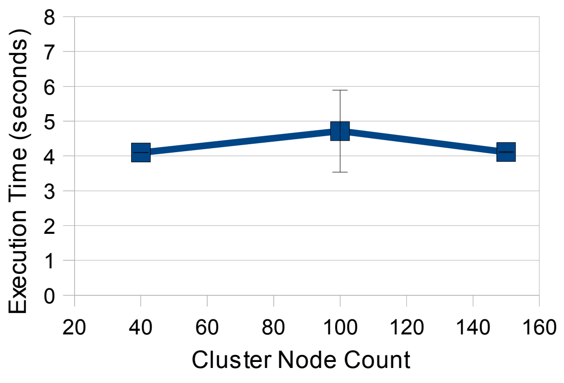

Figure 10: Low volume 3 query mean execution time vs node count (constant size per node)

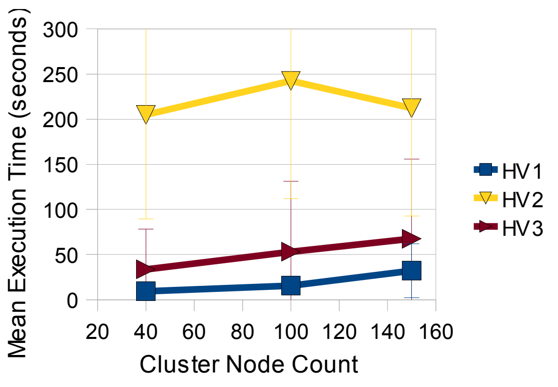

Figure 11: High volume query execution time vs node count (constant size per node)

with the number of chunks since the frontend has a fixed amount of work to do per chunk. Since we varied the set of chunks in order to vary the cluster size, the execution time of HV1 should thus vary linearly with cluster size. HV3 seems to have a similar trend since due to cache effects—its result was cached so execution became more dominated by overhead.

The High Volume 2 query approximately exhibits the flat behavior that would indicate perfect scalability. Caching effects may have clouded the results, but they did not dominate. If the query results were perfectly cached, we expect the overall execution time to be dominated by overhead as in HV1, and this is clearly not the case.

Super High Volume. The tests on expensive queries did not show perfect scalability, but nevertheless, the measurements did show some amount of parallelism. It is unclear why execution in the 100-node configuration was the slowest for both SHV1 and SHV2. Our time-limited access to the cluster did not allow us to repeat executions of these expensive queries and study their performance in better detail. The

6.4 Concurrency

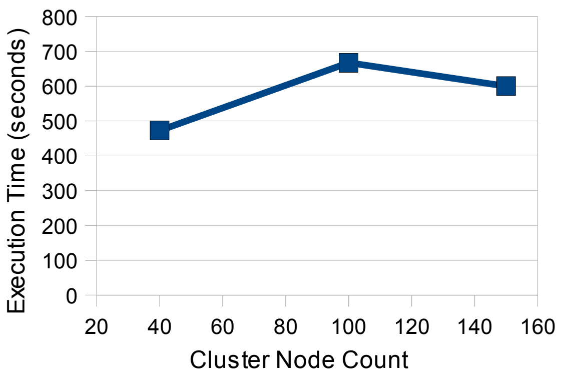

Figure 12: Super High Volume 1 Query Execution time vs node count (constant size per node)

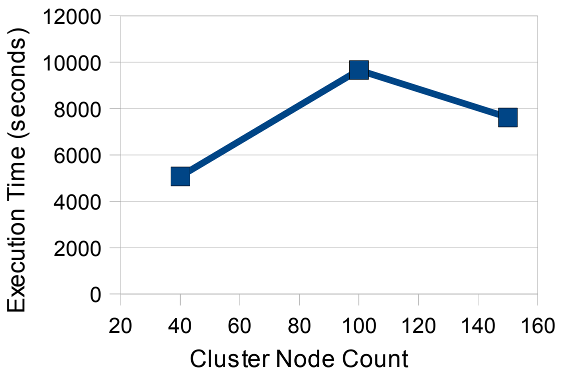

Figure 13: Super High Volume 2 Query Execution time vs node count (constant size per node)

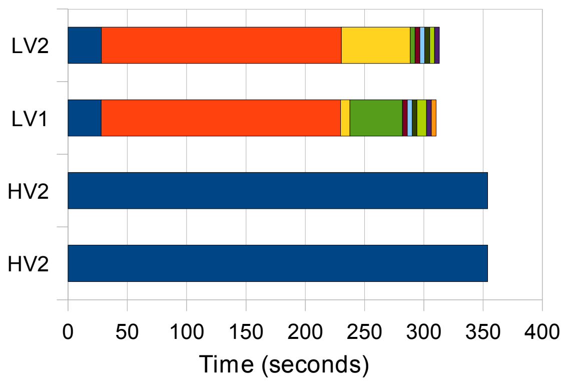

Figure 14: Concurrent execution time for 2xHV2, LV1, LV2 (150 nodes)

We were able to test Qserv with multiple queries in flight. We ran 4 "streams" of queries: two parallel invocations of HV2, one of LV1, and one of LV2. Each low volume stream paused for 1 second between queries.

Figure 14 illustrates concurrent performance. We see that the HV2 queries take about twice the time (5:53.75 and 5:53.71) as they would if running alone. This makes sense since each is a full table scan that is competing for resources and shared scanning has not been implemented. The first queries in the low volume streams execute in about 30 seconds, but each of their second queries seems to get "stuck" in queues. Later queries in the streams finish faster. Since the worker nodes maintain first-in-first-out queues for queries and do not implement any concept of query cost, long queries can easily hog the system. The slowness of low volume queries after the second queries may be curious at first glance, since they should be queued at the end on their assigned worker nodes and thus complete near the end of the HV2 queries. In that case, subsequent queries would land on workers with nearly empty queues and execute immediately. This slowness can be explained by query skew—short queries may land on workers that have or have not finished their work on the high volume queries.

7. DISCUSSION

7.1 Latency

LSST's data access needs include supporting both small, frequent, interactive queries and longer, hour/day-scale queries. We designed Qserv to operate efficiently in both cases to avoid needing multiple systems, which would be costly in development, maintenance, and hardware. Indexing was implemented in order to reduce latency for cheap queries that only touch a small part of the data.

The current Qserv implementation incurs significant overhead in dispatching queries and collecting results. In early development we decided to minimize the intelligence on each worker, so the frontend master became responsible for preparing the SQL queries so that workers did not need to perform parsing or variable substitution. Results collection is somewhat heavyweight as well. MySQL does not provide a method to transfer tables between server instances, so tables are dumped to SQL statements using mysqldump and reloaded on the frontend. This method was chosen to speed prototyping, but its costs in speed, disk, network, and database transactions are strong motivations to explore a more efficient method.

7.2 Solid-state storage

Some of Qserv's design choices (e.g. shared scanning in section 4.3) are motivated by the need to work around poor seek performance characteristics of disks. Solid-state storage has now become a practical alternative to mechanical disk in many applications. While it may be useful for indexes, its current cost differential per unit capacity means that it is still impractical to store bulk data. In the case of flash storage, the most popular solid-state storage technology, shared scanning is still effective in optimizing performance since DRAM is much faster than flash storage and flash still has "seek" penalty characteristics (though it is much better than spinning disk).

7.3 Many core

We expect the performance to be I/O constrained, since the workload is data, not CPU performance limited. It is uncertain that many cores can be leveraged on a single node since they will be sized with only the number of disk spindles that saturate the north bridge. Shared scanning will increase CPU utilization efficiency.

7.4 Columnar RDBMS

We are exploring the use of a columnar RDBMS[3] like MonetDB[1] instead of MySQL, since Qserv is intended to be independent of a particular RDBMS implementation. A columnar organization is likely to speed joins and overall query performance for the wide tables we use, but we have not done sufficient testing yet.

7.5 Alternate partitioning

The rectangular fragmentation in right ascension and declination, while convenient to visualize physically for humans, is problematic due to severe distortion near the poles ($\phi \pm 90°$). We are exploring the use of a hierarchical scheme, such as the hierarchical triangular mesh (HTM)[13] for partitioning and spatial indexing. These schemes can produce partitions with less variation in area, and map spherical points to integer identifiers encoding the points' partitions at many subdivision levels. Interactive queries with very small spatial extent can then be rewritten to operate over a small set of fine partition IDs. If chunks are stored in partition ID order, this may allow I/O to occur at below subchunk granularity without incurring excessive seeks. Another bonus is that mature, well tested, and high-performance open source libraries exist for computing the partition IDs of points and mapping spherical regions to partition ID sets.

7.6 Distributed management

The Qserv system is implemented as a single master with many workers. This approach is reasonable and has performed adequately in testing, but the bottlenecks are clear. A Qserv instance at LSST's planned scale may have a million fragment queries in flight, and while we have plans to optimize the query management code path, managing millions from a single point is likely to be problematic. The test data set described in this paper is partitioned into about 9000 chunks, which means that a launch of even the most trivial full-sky query launches about 9000 chunk queries.

One way to distribute the management load is to launch multiple master instances. This is simple and requires no code changes other than some logic in the MySQL proxy to load-balance between different Qserv masters. Another way is to implement tree-based query management. Instead of managing individual chunk queries, the master would dispatch groups of them to lower-level masters which would could either subdivide and dispatch subgroups or manage the individual chunk queries themselves.

8. CONCLUSION

The LSST query access system makes heavy demands which are not satisfied by any current available software system, and we have therefore implemented a prototype system on top of widely-used database software and distributed filesystem software. This approach should meet all of the stated requirements of user ad-hoc query performance, reliability, incremental scalability, and low cost, at the multi-petabyte scale. Although this system, called Qserv, is still under development, we have illustrated its performance under several types of queries, studying its ability to meet performance goals at a reduced scale as well as its ability to scale to production data sizes.

Online documentation of Qserv:
`http://dev.lsstcorp.org/trac/wiki/dbScalableArch`
Qserv source: `http://dev.lsstcorp.org/trac/browser/DMS/qserv`

9. REFERENCES

[1] P. A. Boncz and M. L. Kersten. Monet: An Impressionist Sketch Of An Advanced Database System. In *Proceedings IEEE Basque International Workshop on Information Technology*. Stichting Mathematisch Centrum, 1995.

[2] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[3] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, SIGMOD '85, pages 268–279, New York, NY, USA, 1985. ACM.

[4] A. Dorigo, P. Elmer, F. Furano, and A. Hanushevsky. XROOTD-A Highly scalable architecture for data access. *WSEAS Transactions on Computers*, 1(4.3), 2005.

[5] Greenplum. http://www.greenplum.com. Retrieved on 2011-06-24.

[6] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 383–394. ACM, 2005.

[7] Hbase. http://hbase.apache.com. Retrieved on 2011-06-24.

[8] Hive. http://hive.apache.com. Retrieved on 2011-06-24.

[9] Z. Ivezic, J. A. Tyson, T. Axelrod, D. Burke, C. F. Claver, K. H. Cook, S. M. Kahn, R. H. Lupton, D. G. Monet, P. A. Pinto, M. A. Strauss, C. W. Stubbs, L. Jones, A. Saha, R. Scranton, C. Smith, and LSST Collaboration. LSST: From Science Drivers To Reference Design And Anticipated Data Products. In *American Astronomical Society Meeting Abstracts #213*, volume 41 of *Bulletin of the American Astronomical Society*, Jan. 2009. http://arxiv.org/abs/0805.2366v2.

[10] S. Kent. Sloan digital sky survey. *Astrophysics and Space Science*, 217(1):27–30, 1994.

[11] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[12] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[13] A. S. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, P. Kukol, and A. Thakar. Indexing the Sphere with the Hierarchical Triangular Mesh. *ArXiv Computer Science e-prints*, Jan. 2007.

[14] W. D. Technologies. *WD RE2 SATA Hard Drive Series Spec Sheet*, June 2008. Retrieved on 2001-06-24.

[15] Teradata. http://www.teradata.com. Retrieved on 2011-06-24.

[16] O. USA. *MySQL 5.1 Reference Manual*, chapter 14. revision: 26533 edition, 2011. http://dev.mysql.com/doc/refman/5.1/en/index.html.

[17] L. Vnuk. SqlSQL2.
http://www.antlr.org/grammar/1057863397080/index.html,
2003. Retrieved from ANTLR site on 2011-06-23.