Market Requirements Document

Feature Name: Parallel Query Engine

Version: 3 Date Submitted: 03/28/05
Version: 2 Date Submitted: 03/07/05
Version: 1 Date Submitted: 01/12/05

Completed By: Leon Guzenda

Description of the Problem

Objectivity/DB has proven itself to be more scalable than both RDBMS and other ODBMSs in many ways:

- Number of concurrent users
- Number of addressable objects and total amount of data stored
- Data throughput

However, ad hoc queries are currently handled synchronously within the thread or process that initiates the query. Microsoft SQL Server beat Objectivity/DB in a test of ad hoc query capabilities at the Sloan Digital Sky Survey project, ultimately leading to the replacement of Objectivity/DB in some parts of that project. The main reason that SQL Server was faster was that it optimized the query and split the search over multiple concurrent threads.

Description of the Requested Feature

This MRD requests that a Parallel Query Engine [PQE] be built for Objectivity/DB. The goal is to dramatically reduce the time it takes to search extremely large (multi-Petabyte) databases, particularly in the absence of any indices to the target objects. If there are any indices then they will be exploited.

The PQE may be started from a process or thread, or it may run as a freestanding server with multiple clients. The controlling process or thread, the Query Manager, will accept a query request in Predicate Query Language syntax and then invoke a user supplied or built-in hook to determine how to split the query across multiple threads.

The standard Query Manager hooks will support the selection of a set of target databases and containers based on:

- A time slicing algorithm, or
- A latitude-longitude square, or
- A class name

The number of service threads will be configurable at the time that the Query Manager is initiated. The Query Manager will pass each service thread a predicate and a single target container identifier. If there are more candidate containers than threads then the Query Manager will martial the service threads according to their availability. The service thread will invoke a user supplied hook as a part of its qualification of objects. There will be a default null hook. There could also be hooks for NexQL and Google searches. The service thread will return one or more qualified OIDs to the Query Manager. The minimum number to be returned may be specified at the time that the Query Manager is started. The service threads will return the OID or OIDs via a shared memory queue controlled by the Query Manager. The iterator in the thread or process that invoked the query will read the object into the local cache.

It should be possible to invoke a hardware search accelerator, such as NexQL, from the Query Manager, rather than standard service threads.

Part of an existing feature or does it require another feature, if so, which one?

This is a new feature of the standard APIs. We may decide to make it a low cost option.

How is this problem being solved now, and why isn't that acceptable?

Users must build their own parallel query servers. They don’t have access to a map of class instances, which could be maintained by our kernel. This feature will make our products more competitive.

What languages must support this capability?

- C++
- Java.
- SQL++

Which platforms must be supported?

- All current platforms, but we could focus on the 64-bit and parallel processor platforms first.

- Real-time platforms probably do not require this capability.

Do any competitors already have this feature?

- DB2, Microsoft SQL Server, Oracle and Sybase.

Customers who require this feature

- VLDB sites.
- Ontology Works (a partner), for their High Performance Knowledge Server product.

Revenue at risk, or which could be won

- We have already lost revenue at Johns Hopkins because of poor ad hoc query performance.

When is this required?

- Release 10 or sooner.

Additional Notes

1. The PQE should be configurable for different environments. Options could include:
    - In process (where we are today, but running in parallel)
    - Multi-threaded
    - Multi-process
    - Local (all on the same host)
    - Remote (distributed manager and/or service agents)
    - Grid enabled

2. We will also need:

    - Marketing collateral
    - Qualification questions/notes for each market sector

- Sales training material
- The attached Technical Note discusses some of the related issues and benefits.

Attachment 1

A PARALLEL QUERY ENGINE FOR OBJECTIVITY/DB

INTRODUCTION

This Technical Note explores the notion of a parallel query engine built into or serving Objectivity/DB. The goal is to dramatically reduce the time it takes to search extremely large (multi-Petabyte) databases, particularly in the absence of any indices to the target objects. If there are any indices then they will be exploited.

APPROACH

The hierarchical storage hierarchy of Objectivity/DB makes it easy to run parallel searches across all or chosen databases or containers. The parallel search engine would be developed and tested in phases:

Phase 1 - A brute force parallel search engine would be started with a number of service threads, a free thread pool, which would be initialized to a list of all of the service threads and (possibly) a master query thread pool. Each parallel search could use some or all of the available service threads. Then:

a) The client (above or within the kernel) would initialize a shared queue[1], used for receiving results asynchronously from the query threads. It would pass the query (i.e. the iterator control information, including predicates) to the query engine, along with a recommended number of threads to be used. The client would then iterate over the results, taking the OIDs from the shared queue, rather than running the mechanics of each step sequentially in the current process/thread.

b) The query engine would iterate over the catalog of databases allocating a DBID (#DB-0-0-0) to each thread, up to the maximum number available or demanded. It would allocate any remaining databases to exhausted threads once the threads had performed their allotted search.

c) Each thread would perform its search on the allocated database and return the OIDs of qualifying objects to the shared queue for this query. Once it had completed its search it would be given a new one to perform, unless there were no more databases to search, in which case it could be returned to the free thread pool. Initially, the search thread would simply call a standard predicate query iterator with a single database as its scope. Or it could call SQL++ with a single database as its RANGE.

d) When all databases have been searched and the master iterator has returned all of the qualifying OIDs to the caller then the master query thread can be returned to a query thread pool, or it can be destroyed.

Phase 2 - A simple refinement of the above approach would allocate one or more containers to each thread, rather than a whole database.

Phase 3 – Many databases have class instances that are unevenly distributed across the federation. An extreme case would be a single instance of an object created in a random container in a random database in a federation of thousands of databases, each having thousands of containers.

The Phase 2 search engine would be supplemented by a kernel enhancement that would build a Class Population Map [CPM]. The CPM could consist of a hash table (one per federation) of type-to-container clues. Each entry (clue) in the hash table would consist of the concatenation of Class Type Number (TypeID) and OCID, where the OCID [#DB-OC-0-0] identifies an OCluster that most probably (or perhaps certainly) has at least one instance of an object of that TypeID within it.

It will be far too expensive to update a shared hash table every time a new object is created, so the clues will be inserted into the hash table the first time that a transaction creates its first instance of a class in a

---

[1] The shared queue mechanism would be a useful product in its own right, particularly for telecom applications and servicing event notification classes.

given container. This “insert” probably has to be a delayed (current process/thread) action, to avoid having to keep the hash table open for update for the duration of the transaction.

The clues will be linked together in lists headed by a single TypeID entry (with OCID=#0-0-0-0). The clue header (or an associated structure) should contain a reference count of the number of instances of that type in the whole federation. This makes it possible for a master search iterator to rapidly locate the containers that have instances of the class targeted by the query. It can hash to a clue in the CPM and chain down the list of target containers, making it very efficient.

There is one interesting issue. If a query that is creating or deleting data wishes to run a parallel query then the master query iterator must look at the committed CPM entries and merge them with the uncommitted entries. If the master query iterator is running synchronously in the current process/thread then this is no problem. If it is a separate thread/process then executing a query must also deal with uncommitted data.

Note that this technique could be tested without making kernel changes by building a standalone application that scans all containers and builds the CPM. This would work for read-only databases and might be a useful tool in its own right.

ANTICIPATED PERFORMANCE IMPACT

Case 1 [Best Case] VLDB with many databases and containers and no updaters.

Consider a 100 Terabyte federation, consisting of 50,000 databases, each with an average of 1000 containers and with the database files averaging 2 Gb. Assume that there is exactly one instance of the target (unindexed) object in an unknown container, so all containers must be searched. The current algorithm would perform:

- 50000 database lookups (locking them and reading the file housekeeping)
- 50,000,000 container lookups (which would incur at least that many I/O operations).
- Approx. 12,500,000,000 data page I/Os (assuming 2 Gb files, 8kb page size).

The total is in excess of 12.5 billion I/Os, which at 100 I/Os per second would take about **4 years** to run.

The new algorithm would require:

- 1 lookup into the CPM
- 1 database lookup
- 1 container lookup
- 1 to 250 data page reads.

At worst, at 100 pages per second, this would take **under 3 seconds**. At best, it would be practically instantaneous.

The downside is the overhead of maintaining the CPM, which imposes a noticeable burden the first time that an instance of a given type is created in a container used by the current transaction. If the database has to sustain heavy query traffic then this overhead is justifiable.

Case 2 [Bad Case] A single database, not many containers and many concurrent updaters.

Consider a federation with one frequently updated database (2 Gb) with 1000 containers, heavy query usage, a single (unindexed) target instance in an unknown container, and a large number of concurrent updaters.

The current algorithm would require:

- 1 database lookup
- 1000 container lookups
- 1 to 250 data page lookups (assuming 8 kb pages).

At worst, at 100 pages per second, this should take **less than 13 seconds**. In fact, it probably takes more because of current catalog inefficiencies (to be removed at Release 9), but let's assume that the new catalog will make this overhead less noticeable. At best, the query would be practically instantaneous.

The new algorithm would require exactly the same number of operations as the best-case scenario, described above, i.e. it will take **less than 3 seconds**.

The downside is that updaters may be very heavily impacted because the CPM has to be updated, adding an additional container update to every transaction. In this case the designers would probably opt to stick with sequential query processing.

Case 3 [Typical Case] VLDB with many databases and containers, random updaters and many qualified objects.

Consider a 100 Terabyte federation, consisting of 50,000 databases, each with an average of 1000 containers and with the database files averaging 2 Gb. Assume that there are many instances of the target (unindexed) object in unknown containers, so all containers must be searched. The current algorithm would perform:

- 50000 database lookups (locking them and reading the file housekeeping)
- 50,000,000 container lookups (which would incur at least that many I/O operations).
- Approx. 12,500,000,000 data page I/Os (assuming 2 Gb files, 8kb page size).

The total is in excess of 12.5 billion I/Os, which at 100 I/Os per second would take about **4 years** to run.

The new algorithm would require:
- 1 lookup into the CPM
- N lookups into containers that are known to contain qualified objects
- 1 to 250 data page reads per container.

At worst, at 100 pages per second, the new algorithm would take:

Number of containers involved	**Search Time (Seconds)**		
	Sequential Search (1 thread)	**Parallel Search** (100 threads)	**Parallel Search** (1000 threads)
	~	~3	~3
1(	~25	3+	~3
1,0(	~250	25+	~3
10,0(	~25,00	250+	25+
100,0(	~250,00	2500+	250+
1,000,0(	~2,500,00	25,000+	2,500+

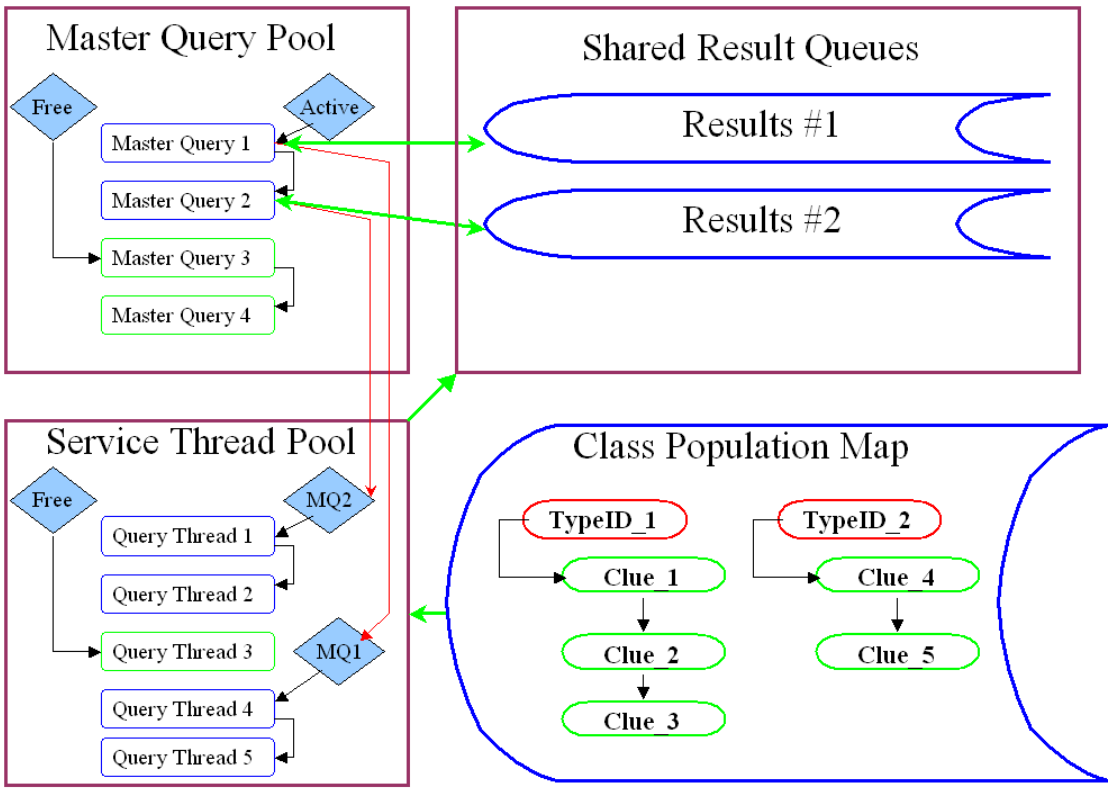
These figures assume that everything is online, that there is enough hardware to allow parallel I/Os and that the receiving program takes almost no time to deal with each qualified object. In practice, the marshalling

of objects via the shared queue would impose some minor CPU overheads. However, in the case where 1,000,000 containers (2% of the total) hold qualifying objects and 1000 threads can be used effectively the **Objectivity/DB search time would probably drop from 4 years to under an hour.**

PQE STRUCTURE

Notes:

1. The Master Query Pool manages the Master Query Iterator threads.
2. Each Master Query Iterator initializes a Shared Results Queue and controls one or more Query [Service] Threads. It broadcasts a container ID and the search predicate string to each Query Thread.
3. The Query Threads are managed by a Service Thread Pool.
4. The QueryThreads perform a search through a particular container, using information from the Class Population Map.
5. The Query Threads write results (OIDs) back to a Shared Result Queue.
6. The Master Query Iterator takes its results from its Shared Result Queue.



...document continues...

BUILDING A PROTOTYPE USING JMS

I believe that we can build a prototype PQE quite quickly by leveraging the existing ooSession class and the Java Message Service [JMS] Point-to-Point [P2P] protocol. The P2P protocol has the following purpose:

[Quote[2]…] P2P messaging is designed for use in a one-to-one delivery of messages. An application developer should use P2P messaging when every message must be successfully processed. Unlike the Pub/Sub messaging model, P2P messages are always delivered. Some of the characteristics of the P2P messaging model are as follows.

- *Uses message queues, senders, and receivers.*
- *Messages are sent to specific queues; clients will extract messages from specific queues established to hold their messages.*
- *Queues retain all messages sent until such time as the messages are consumed or expire.*
- *Each message has only one consumer, though multiple receivers may connect to the queue. [Leon: This is not needed by PQE, at least initially]*
- *Messages are removed from the start of the queue (FIFO).*
- *There is no time dependency -- a receiver may acquire a message whether of not the service was available at the time the message was sent.*
- *Receivers acknowledge the successful receipt of a message.*

[…End Quote]

This diagram[3] shows a typical JMS P2P setup.

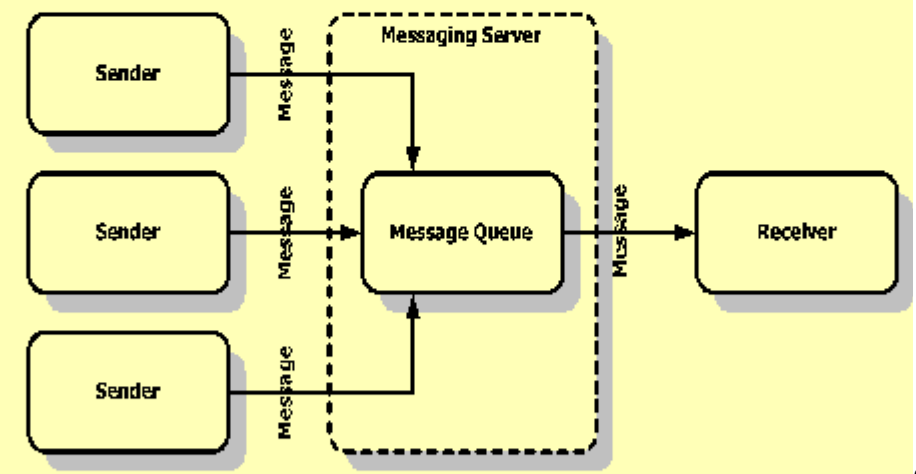


Figure 2: Queue Based (Point-to-Point) Messaging System

The PQE clie PQE Query Thread ey must also send an "

[2] Published by T.A. Flores at http://www.onjava.com/lpt/a/809

[3] Published by Gopalan Suresh Raj at http://my.execpc.com/~gopalan/jms/jms.html