Objectivity Inc.

Enhanced Index Management MRD

Version 1.0

Revision History

Date	Version	Description	Author
11/23/2008	1.0	Initial	Lenny Hoffman

Table of Contents

MRD

1.Introduction

The purpose of this document is to collect, analyze and define high-level needs and features of the *Enhanced Index Management* project. It focuses on the capabilities needed by the stakeholders, and the target users, and why these needs exist. The details of how the *Enhanced Index Management* project fulfils these needs are detailed in the solftware requirements document (SRD).

1.1.Definitions, Acronyms and Abbreviations

1.2.References

1.3.Overview

Until recently with the Enhanced Object Qualification project, query support has remained virtually untouched in Objectivity since its introduction in version 2.0 back in 1992. Since then interest in query among prospects and customers has steadily increased from almost zero, when most customers were using us as a mere persistence engine, to almost 100%, as most customers look for us to provide more traditional database features like query, especially as their database sizes have increased.

Back in the early 1990's our biggest markets where Engineering, Telecommunications, Process and Control, and Scientific. The applications in these markets were heavily focused on complex data accessed navigationally. When they did run into a query need, it was limited such that they could use the 2.0 solution, build their own query support, or use a relational DB on the side.

It became common knowledge in our markets, and indeed the entire OODB industry, that OODBs were not good at ad-hoc queries and that relational DBs were. Certainly, our query support did nothing to challenge that view. Interestingly, from a core technology vantage point the opposite should be true; Objectivity can be configured to look like and act like a relational one (thus be queried like one), but where relational DBs are handicapped by having to obey the relational model, queries in Objectivity can be further supported by flexible object placement and direct support of relationships (pointers).

Objectivity's storage hierarchy enables users to choose their own primary organizations, which has turned out to be a most powerful feature. Take for example an application that stores events that all have a time stamp and lookups almost always include a time component. This application can have DBs represent years and containers weeks, and maintain a little data structure (let's call it a user defined index) that keeps track of them. Then, when looking up events with at least a time component, the index can be used to narrow the search from the entire FD of events to a particular container.

Unfortunately the current query support does not recognize user defined indexes of any sort, those that resolve to scopes or target objects, so users have to first use their indexes and then delegate to scan once scopes are identified.

Since indexes are collections used to improve performance of queries, this project is mostly about performance, but ease of use is also targeted.

2.Feature Requirements

2.1.Problems

There are several problem areas:

- Index scalability
- Index type limitations
- Index utilization

- Duplicate collections
- Index update
- Index key description problems
- Index use visibility

2.1.1. Index scalability problems

An index cannot grow in size beyond a single container.

This is a real problem for those of our customers who collect large amounts of data that they must later search quickly (e.g. our intelligence customers).

2.1.2. Index underutilization problems

No support for indexes that resolve to databases or containers

Our current single index structure is relegated to resolving to individual objects, but in many applications there are collections of databases or containers that organize objects based on type or values such as timestamps. Not having a way for them to be used by the query system means that users have to implement their own query optimizer that knows how to utilize their index when possible, or simply they don't get used to optimize a query.

Indexes at enclosed scopes are never used

Currently, a scan performed at FD scope only considers indexes registered at that scope, and if none exist or none apply to the given predicate, then all pages in the entire FS are scanned, even when applicable indexes exist in enclosed DBs or containers. Similarly, a scan performed at DB scope does not consider indexes at container scope. This is a gross missed opportunity for query optimization, especially when combined with P2 as it is a common pattern to have a collection of containers which have indexes.

Indexes are not used without keys

Currently an index is only used when attributes in the predicate are supported by the index. This is another gross missed opportunity for query optimization; because objects of a given type can be spread about, scanning the index collection to get directly to them will often be much better than scanning all pages in the query scope.

2.1.3. Index type limitation problems

Only one collection type available

Currently, the only collection available for use as an index is a B+ tree. The problem is that this is not always the best performing choice. A few examples:

- A simple list is better in cases where objects are looked up and used without applying a filter, as objects of the right type are found without a scan just as with any other index, but the performance hit if updating the index with every object

edit is eliminated. Note that relational databases have little use for this type of index, as tables provide this access, but with our independence between type and placement, objects of a given type can be spread out and having the ability to get directly to them as a set can represent a significant performance improvement.

- A hash based collection can be a lower overhead option to tree when range base lookups are not needed.
- Quad tree's and other multidimensional index structures are usually better when more than one attribute is being indexed on.

No support for Indexes that are user defined collections.

Specialized index structures for specialized situations are commonly the secret sauce of applications that allow them to perform especially well. Not having a way for them to be used by the query system means that users have to implement their own query optimizer that knows how to utilize their index when possible and they also have to make sure that the index is kept up to date with changes to indexed objects. Neither is a simple task and together constitutes a serious ease of use issue.

2.1.4. Index collection and index registry accessability problems.

Index collections not accessable as collections

Our current index structure is private to scan; there is no public API with which users can directly utilize it as a collection. Applications often take advantage of both declarative and programmatic queries, the former facilitating ease of use and ad-hoc, while the latter facilitating optimal performance for specific queries. With the index private to scan combined with P6, such applications end up having to have two collections. There are several problems with this:

- An index can take a lot of space, having a duplicate collection means doubling disk consumption.
- Update performance is worsened by having two collections to keep up to date with changes.
- Unlike with the scan specific collection, the application is responsible for keeping the application accessible collection up to date, which is an ease of use problem.

Index registration not public API

The registry of indexes at a scope (container, DB or FD) is private to scan and ooKeyDesc.

In order to be able to use a collection both declaratively and programmatically, the registry needs to be available for interrogation; otherwise users must maintain their own reference to the collection for when they want to access it programmatically.

In order to be able to register a collection at a scope when it has already been created the registry must be available.

2.1.5. Index update problems

Indexes have to update themselves without knowledge of previous key values.

For indexes that are based on keys, at index update time the entry for the original key value must be looked up so that it can be removed prior to inserting a new entry for the new key value. The problem is that with the current index update mechanism, the original key value is not known, thus the entry for the original key value must be searched for based on OID and since the collection is optimized for key lookup not OID lookup, that can be a very slow operation.

Our current index deals with the slow lookup by OID by utilizing a second collection organized by OID that points into the first.

Along with being a general performance problem, this also complicates P5 and P6, as with each additional collection type to be utilizable as an index, the lookup by OID problem exists, which either means that index update suffers greatly because a second collection or some other mechanism does not exist, or ease of use suffers with the requirement of having to design for fast lookup by OID in addition to fast lookup by key.

While with not having the original key value having the second collection is better than not having it, it is still costly compared to being able to have a single collection, as two collections take up more disk space, require more IO and creates an index insert performance hit, as two collections have to be updated instead of one.

Indexes are asked to update themselves even when they don't require an update

The current mechanism only knows that a page has been marked dirty, not which existing objects were changed or that those changes affect an index. The algorithm is:

For each object on the page:

- Its type is determined and the list of indexes in the current scope is determined based on that type.
- For each index in that list:
  - It is determined if the object is new or existing.
  - If new:
    - It is inserted in the index.
  - If existing:
    - Its entry in the index is found and a comparison is made to determine if the index requires updating.
    - If it requires updating:
      - The existing entry is removed.
      - A new entry is inserted.
    - If it does not:
      - Do nothing

Note that indexes are updated immediately upon an object being deleted.

The problem is the (do nothing) above takes time to get to (looking up entries in an index is a costly operation) and can happen a lot. One case is when a small percentage of the page's existing objects are

updated. Another is when updates on objects do not affect indexes, which turns out to be quite common, as most indexes are created on mostly or completely immutable data, such as names or IDs. This constitutes a significant performance hit.

Indexes at FD or DB scope are updated using costly micro transactions

This is done to minimize lock contention, but micro transactions incur much lock server communication and journal file work.

It is this overhead that has made FD and DB scoped indexes much less practical than those scoped to a container.

An SE or consultant led optimization usually results in some type of application maintained global collection that leads to containers that then have indexes, but because of P6, P2, and P3, this workaround means that utilization of this global collection must be managed by the application.

2.1.6. Index key description problems

Index key descriptions are scattered about

There is no standard place for placing ooKeyDesc objects when using the C++ binding and it is up to the user to decide where to place them. Along with being another task to learn about and perform, where these objects are placed has a performance impact. When an index is used its key description is accessed, so if the key description object is in a different container than the index, then an additional container must be opened if not already open. The Java binding solves the ease of use issue by placing key description objects for users, but it does so in a way that practically guarantees that a separate container must be accessed. In any binding, for DB and FD scoped indexes, there is no way to place the key description object in the same container as the index.

ooKeyDesc cannot describe key only data.

All of the current key descriptor's fields refer to attributes on the indexed class, there is no way to describe the situation where data exists in the index, but not on the indexed class.

It is a popular choice to have data only in the index in those cases where access to indexed objects is always through the index as it saves having that data both in the index and in the indexed objects. Take for example an index whose key contains a date and leads to containers organized by day. Having day be included in all events stored in those containers would incur a great deal of unnecessary overhead.

ooKeyDesc is not a part of schema, thus it cannot be determined which shema attributes have been indexed.

This information is nice to know for users, such as those looking for such information via Assist or the PD, but it is also problematic for solving P10, as it makes determining if an update affects or can affect an index very difficult. Note that searching the FD for all ooKeyDesc instances to build up this information is simply not practical.

2.1.7. Index use visibility problems

It is not visible when indexes are used

There is no way to tell if scan uses an index or not, and if so, which one. This causes difficulties for those attempting to tune performance by the addition, modification or removal of indexes.

2.2.Goals

Have scalable indexes.

Ours database is meant to be scalable, but when users encounter any aspect that is not without reasonable scalable alternatives, then their perception becomes that it is not scalable. We need to be fully true to our scalability story, including indexes.

This means solving PROBLEM17.

Utilize available indexes.

If an index exists that can help speed up a query, then it should be used.

This means solving PROBLEM18, PROBLEM19 and PROBLEM20.

Allow collections of any collection type to be utilized as an index.

Different types of collections serve different purposes better, thus to allow for the best possible performance, we need to support any type of collection, including those defined by the user for their unique lookup needs.

This means solving PROBLEM21 and PROBLEM22 so that a collection of any collection type can be an index.

Solving this effectively such that any collection type is not merely allowed, but also easily introduced, means also solving PROBLEM25. Without solving PROBLEM25 means that lookup by OID has to be solved for each new collection type introduced.

Support indexes with index only data.

It is not always necessary to have an index's key contain copies of data existing on indexed objects, and when not it is a performance advantage, by reducing overall IO, to only have the data in the index to avoid having two copies.

It is not necessary to store key data in the indexed objects is when access is always through the index. For example, if looking up a person by name is the first thing done for working with that person, then storing that value in the person object is not necessary.

This means solving PROBLEM29.

Improve index update performance.

While indexes generally improve query performance, they do so at some update performance cost. We

need to minimize that update cost to attain the best overall performance possible.

Do so by solving PROBLEM25, PROBLEM26, PROBLEM27, PROBLEM28 and PROBLEM30.

Eliminate any need to have duplicate collections.

Duplicate collections worsen performance and ease of use.

This means solving PROBLEM23.

Enable directly adding a collection as an index at a scope.

The collection may have already been created for other uses and later added as an index at a scope to take advantage of its ability to speed up declarative queries. This is likely to happen a lot for existing customers who already have their own custom indexes, but up until this project they were not available to scan, but with this project they now want to make available.

This means solving PROBLEM24.

Provide visibility to index use.

We need this to enable those tuning performance to do their job most effectively.

Do so by solving PROBLEM31.

3.Relationship to other features

3.1.Part of an optional feature?

No.

3.2.Other features required.

First Class Collections, for collection and key support.

3.3.Other features requiring this one.

All declarative lookup features such as the current PQL and SQL and future ones such as LINQ benefit from improved lookup performance.

3.4.Other features that work with this one.

The placement manager provides indexes that lead to containers based on type and sometimes key values as well.

4.Positioning

4.1.Competitors that have this capability:

Don’t know.

4.2.Customers that require this capability:

Customers who use scan or our SQL interface and use, or wish to use, indexes to improve performance.

Customers who have not been able to use scan due to it not taking advantage of their custom indexes.

4.3.Revenue at risk or which could be won:

Indexing is a well known and expected means of improving query performance; since our discriminating values that lead to winning new business are mostly about performance, we need that to be the case for query, not just navigation as in the past. With improved query performance we can win a larger portion of the current and emerging crop of applications that have outgrown their Relational database by providing them equivalent or better query performance for what they are doing now in addition to our other advantages.

4.4.When is this required?

This is required as soon as possible.

5.Extent.

5.1.Languages that must support this capability:

All ULB languages.

5.2.Platforms that must support this capability:

All.

6.Applicable Standards

None.