

Request for Comments

Requirements for Indexing Prototype in HDF5

Mike Folk, Quincey Koziol, James Laird

February 2005

Version 1.3

1. Purpose

The purpose of this paper is to identify the range of uses of indexing by HDF5 applications, to identify a subset of these uses for implementation in a prototype indexing data model, and to sketch requirements for these use cases. One constraint is that the prototype implementation should be of value for working with time-sequenced instrument data.

This work is being carried out by the project “Capturing and Accessing Complex Time-Sequenced Instrument Data With HDF5,” a project of The National Center for Advanced Secure Systems Research (NCASSR) which is funded by the Office of Naval Research (ONR) and is led by the National Center for Supercomputing Applications (NCSA).

2. Background

2.1 What do we mean by indexing?

An *index* is a data structure that allows us to quickly identify and locate objects (data elements or file objects) according to some criterion. An index may have one or more *key fields* containing key values to which search criteria are applied. Alternatively, the index may have an implicit key, which is the location of the object of interest.

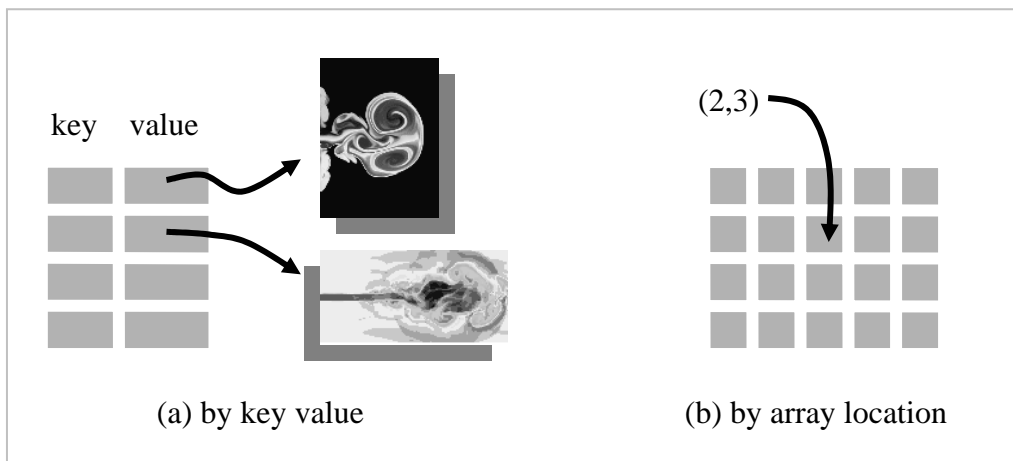


Figure 1. Two interpretations of indexing. (a) Target identified by key value. (b) Target identified by its location.

An *index record* consists of an optional key field, and one or more *value* fields. The value field may contain actual data, or pointers to data or objects of interest.

An *index query* is an operation that describes and applies the search criteria used to locate

an index record. A query is specified in terms of one or more keys, and returns table records.

An *indexed file* is one for which one or more data structures (indexes) exist for finding records or other objects of interest in a file. Indexes can be part of a file, or they can be in a separate file, part of a database, or elsewhere.

Indexes are useful when there is no easy way to know the location of objects that we might be searching for.

As an example, suppose data is collected at varying time intervals from a number of instruments, one record per time interval. Each record includes a time-stamp field, together with measurements from some of the instruments. Each such record is variable in length. Suppose it is also known that applications will require random access to records based on the times when measurements were made. If the records are stored end-to-end in the file, then direct access is not possible. The records are variable-length, so the offset of a given record cannot be computed. This is a good case for building an index. A second, fixed-length record dataset could be created, wherein each record has two fields, the first field containing the time stamps, the second containing the byte offsets of the corresponding records. If this file were sorted based on the values in the time stamp field, it would be relatively easy to do a binary search on this file for the time stamp of interest.

There can be more than one index for a file, providing alternate views. For instance, in the example there could be one index that is sorted by time stamp and other indexes sorted according to other criteria. Each index provides a different view of the ordering of the records, even though the records themselves are ordered in just one way, and hence facilitate quick random access to records.

2.2 Multidimensional indexes

A great deal of data stored in HDF5 is multidimensional, such as images, unstructured grids, and geospatial features. It would be extremely valuable to support indexed access to objects in higher dimension space. This kind of access is out of scope for the current project, but is of interest in the longer term.

2.3 Reasons for a standard HDF5 indexing API

There are a number of reasons it would be useful to provide an open source, centrally supported indexing API for HDF5.

- **Avoid duplicated effort.** Since indexing is otherwise likely to be implemented by many HDF5 users, such an implementation would let users avoid the expense of creating duplicate implementations.
- **Standardize treatment of indexing in HDF5.** Having a standard API increases the chances that indexes will be compatible with one another. Other applications, and tools such as HDFView, can use such indexes to access data in the same ways that the data producer intended.
- **Make indexes portable.** If an index is stored in (or with) an HDF5 file, it can be moved from one platform to another (e.g. UNIX to Windows), be changed on the

new platform, and still be compatible with both systems.

Of course, there are often good reasons not to index datasets in HDF5, in particular when technologies such as database management systems can do the job better. This is addressed in the next section.

2.4 HDF5 indexing and RDBMS

Most of the indexing capabilities that can be added to HDF5 can be done, usually better, with relational database management systems. Indexing in HDF5 is not intended to replace the use of relational or other databases. Rather, it is intended for those situations in which a relational database may be inadequate or difficult to use. These include:

- **Packaging indexes with the data.** Cases where it is desirable to package indexes in the same container as the data. Examples include long-term archiving of data, and transportation of data among different computing environments.
- **Inconvenience of DBMS.** Cases in which the uses of the data are such that the extra overhead of a DBMS is unnecessary; it more convenient for an application to use the simple capabilities provided by HDF5 indexes than to create and maintain a separate DBMS. In fact, one would often pay a heavy price by using an RDBMS just to get indexing if transactions, full SQL, etc., were not needed.
- **Open format.** Database files are often proprietary and awkward to share, HDF5 is much easier to share.
- **Cross platform.** Database files are often not portable across platforms. HDF5 is much easier to share between apps. and platforms
- **Access modes.** With some exceptions, a DBMS alone does not provide the types of data access that are needed in many scientific applications, such as subsetting and parallel access. One could link HDF5 with a DBMS to achieve this, and in some cases this will be desirable, but in others it may require much more work than just using indexing with HDF5.
- **Single format.** Cases in which it is desired to have all the information in the HDF5 format, rather than having some of it in a DBMS format and some of it in HDF5.
- **Scalability.** Cases in which the size or speed capabilities of HDF5 are better than those of a DBMS, and these capabilities are important to an application.

2.5 Scope of current study

The range of possible indexing methods that would be of value to HDF5 users is enormous, and could include everything from simple sorted tables to sophisticated data structures such as B-trees and R-trees. In the current study, the goal is simply to explore a tiny part of that range, but a part that has high payoff for important applications of HDF5. The study will look for ways to do indexing in HDF5 by means of simple table structures whose fields are limited to HDF5 atomic datatypes and fixed-length and variable-length datatypes.

At the same time, understanding the tremendous potential for indexing, a second goal of the study will be to provide a framework for future features in HDF5 that can embrace more sophisticated structures, such as spatial indexes and hierarchical indexes (indexes of indexes).

2.6 Related work

Several forms of indexing have been implemented by HDF users almost since its

beginning. Here are some examples.

HDF5 datatypes and datasets. Indexes occur in a number of instances in HDF5:

- Dataset selections. Selections of array elements in a dataset are a form of implicit indexing.
- Reference datatypes. Reference datatypes identify objects or selections in a dataset. This provides a mechanism for specifying the locations of objects identified by an index.
- Variable length datatypes, as implemented. VL datatypes are implemented in HDF5 by means of an index. The dataset or attribute whose values are variable length datatypes represent these values as pointers into a heap, where the actual values reside.

JPL 1992. In 1992, a team at the Jet Propulsion Laboratory studying the relationship between cloud properties and other environmental parameters used HDF Vdatas to access spatial measure measurement by position and by time.

HDF-EOS point data. The HDF-EOS standard format supports three “earth science data types,” which are used for organizing most of the data collected in NASA’s Earth Observing system (EOS). HDF-EOS uses indexes in a variety of ways, one of which is in its organization of *point data*. Point data is commonly used for data or data records with no ordered spatial or temporal organization, such as *in situ* meteorological observations from weather stations, or ship/buoy reports over the oceans. The data structure for is essentially an index that is searched to find a point of interest.

Table1

Lat	Long	Buoy ID
25.2645	-91.2564	0126
22.3549	-93.4657	3564
23.2564	-89.2546	1256

Table2

Buoy ID	Time	Wave Height	Temp(C)
0126	01:26	2.54	18.4
0126	05:56	3.58	18.2
3564	06:28	12.64	16.4
1256	08:12	7.58	17.1
1256	09:58	7.76	17.2
0126	09:59	4.23	20.1
3564	10:16	10.23	17.5

Figure 2. HDF-EOS point data example, showing index for locating buoys based on latitude and longitude.

PyTables. PyTables is a database package built on top of HDF5, providing efficient access to very large amounts of tabular data. PyTables offer selection method called *indexed* mode, which creates an index based on the values in a particular column in your table. The index can then be searched using a binary search.

HDF5 table API. The HDF5 Table API defines a standard storage for HDF5 datasets that are intended to be interpreted as tables. A table is defined as a collection of records whose values are stored in fixed-length fields, and hence is a candidate for a simple

index.

HDF5 internal B-tree structures. HDF5 uses B-tree indexes very heavily, but only internally. Serious thought has been given to exposing these structures for use as indexes into user data.

3. Use cases

A number of use cases have been identified for indexes. Of those listed below, the first is the driver for this project. However, the others represent uses of indexing in HDF5 that would seem to be quite plausible and valuable. In this project, we should keep them in mind and try not to preclude them in our design.

3.1 *Test data from raw, real-time and/or embedded data systems*

The proliferation of sensors and other instruments introduces enormous challenges to data management. Even for a single event, incoming synchronized time-sequenced data can have many sources. The number of incoming data streams, as well as the types of data, can be large. With flight test data applications, for instance, it is necessary to accommodate data from test aircraft, voice communications, video, ground, satellite tracking, and other sources. Virtually all of this data must be gathered, integrated, processed, visualized, and archived.

3.1.1. Packet data

This type of data is typically collected in the form of data records arriving in a continuous stream from sensors. In some cases the number of sensors can be very large. The data typically arrive in *packets*, which are sometimes very highly structured, but can also be very heterogeneous and be of irregular content and variable in length.

Packets are stored in the order they arrive. They may arrive from different sources, so there is likely to be a merging operation as they are stored. When merging packets, it will not always be possible to store them in the exact time order of collection.

Since packet data arrives in a real-time stream, there is little time to index the data on a global scale, but it is often desirable to be able to look at small subsets of this data – to be able to look back a few packets or forward a few packets. In this case, it might be useful to create indexes on the fly that provide access to data within a local time-frame. It has been suggested that a ring buffer be used to hold the latest *n* records so that an index could be constructed for the recent past. A ring buffer could also be used to sort records into time order before adding them to the file. More than one index may also be desired, if local searches need to occur on different parameters.

3.1.2. Time history data

After collection, the data may be reorganized for post-processing according to some criterion, often as a *time history*. Once the data is archived and organized, the data may be queried and viewed in a wider variety of ways. Whereas packet data presents essentially a per-record, or “horizontal,” view of the data, post processing is more likely to involve per-variable, or “vertical” views of the data. One may be interested in a certain subset of

fields, perhaps those that meet certain criteria. In such cases, it may be useful to create indexes over the entire time history based on specific vertical fields or combinations thereof.

3.2 *In-situ data*

In situ data consists of measurements made at the actual location of an object. Examples of in situ measurements are measurements made by weather balloons and buoys deployed in oceans to make scientific measurements. The common organizing index for in-situ data is one that supports search by object location, but other indexes, such as by parameter values or time, are also of interest. An HDF-based model and API already exist for creating indexes for working with remote sensed *in situ* data: the HDF-EOS point data.

3.3 *Data subset referencing in imagery*

In many cases, data users need to be able to identify particular data elements within images. This occurs across a wide spectrum of application domains, such as earth sciences, medical applications and astronomy.

For example a climate researcher might identify MODIS pixels belonging to a hurricane in a file. These pixels define an irregular object of interest, a “hurricane object”. HDF5 contains a datatype called a “region reference” that can contain this type of information.

In later processing, it is not unusual to want to access just the pixels that belong to these objects, without having to resample the original images. For instance, a scientist might want to compute average values for some parameter in a collection of hurricane objects. If these objects can be accessed directly, a great deal of time and effort can be avoided. Indeed, the scientist may not even be able to extract the original object.

These collections of references constitute index structures in which one of the fields is a region reference, identifying the pixels that constitute the original object, and other fields contain search information and other metadata.

3.4 *Group-ordering*

Sometimes it is important to organize at the object level with a file or other collection. HDF5 provide a grouping structure for this, but does not provide a way for users to define criteria for organizing items in a group.

Some applications, such as netCDF and CGNS, need to refer to object according to when they were added to a group. An index structure might provide this capability.

As another example, an application may need to store a sequence of 10,000 images in an HDF5 group, as an animation, and to order these images as frames in an animation. An index could be used for this. If the animation contained certain subsequences depicting events of interest, a second index could be created to identify the beginning frame of each event.

3.5 *Color lookup table (palette)*

A color lookup table is a simple form of index. HDF5 currently defines a palette as part

of its high level image API, described in the “HDF5 Image and Palette Specification” (<http://hdf.ncsa.uiuc.edu/HDF5/doc/ADGuide/ImageSpec.html>). It may ultimately be desirable to re-define this in terms of a more general HDF5 index.

3.6 Connectivity list

Another form of index commonly used in HDF5 is the connectivity list, which identifies the vertices, and their order of connectivity, of polygons or polyhedra in an unstructured grid. The values in a connectivity list are the indices of the nodes in an array of nodes. Often a connectivity list is stored as a 2-dimensional array, in which a row of the array corresponds to a polygon or polyhedron, and the elements on the row contain the vertices.

3.7 Spatial feature list

Another form of index is a geospatial feature list. Geospatial features are commonly described using vectors of points in two or three dimensions. The geospatial features in a collection can vary greatly in terms of the number of points used to describe them. These are often stored in a 1-dimensional array of points, with feature index describing the offset and length of the corresponding feature.

3.8 Spatial indexing

The number of possible uses for spatial indexing in HDF5 is very large. As mentioned in the introduction, it is out of scope for the current study, except that any framework created for indexing should accommodate later support for spatial indexing, if possible.

There is one form of spatial indexing that is supported by HDF5, the region reference. A region reference describes a region within an HDF5 dataset. An index could be created in which dataset regions were the data elements.

3.9 DNA sequences

Some users are beginning to use HDF5 to store DNA sequences. It is often necessary to find sequences according to some particular criteria, and perhaps indexing could provide a fast way to do this. At the same time, it is not clear what kind of indexing would enable what kind of operations on DNA sequences. Most bioinformatics applications use string algorithms like BLAST rather than indexed operations.

3.10 SRB and indexing

The SRB (Storage Resource Broker, <http://www.sdsc.edu/srb/>) from the San Diego Supercomputing Center is client-server middleware that provides a uniform interface for connecting to heterogeneous data resources over a network and accessing replicated data sets. SRB, in conjunction with the Metadata Catalog (MCAT), provides a way to access data sets and resources based on their attributes and/or logical names rather than their names or physical locations.

There are at least two cases in which indexing can be very useful in connection with the SRB.

1. When an HDF file is uploaded to SRB server, indexing can be used to ingest the file information into MCAT. Later, file information can be retrieved

through filtering and searching from MCAT without accessing making direct calls to the HDF5 library.

2. Sending the entire index to an SRB client once can save significant network transportation time between the client and server.

3.11 Database archiving

The scalability and structural richness HDF5 makes it a candidate for database archiving, particularly for databases of very large objects. Relational tables could be stored in HDF5 as HDF5 indexes, with other HDF5 structures used to store complex objects. On the other hand, a typical use case for database archiving is to write the whole database out, which is a sequential operation, and then on restore to read it all in, which is a sequential operation, and then access it using an RDBMS's indexing and API. Since RDBMS applications are typically heavily optimized for the RDBMS's indexing algorithms, it's unclear what replacing them with simpler algorithms accomplishes. Other use cases, such as a partial restoration, may be more appropriate. This use case category needs clarification.

4. Requirements for primary use case

Given the limited scope of this study, the prototype can cover only some of the use cases described. Among the use cases, one common set of simple functions would seem to be of value: a simple index for selections in datasets that that supports simple queries on a number of fields. Such an index suffices for those situations in which it is not necessary to update the index dynamically.

Such a simple covers the following use cases at least to an extent that should be useful for a significant portion of applications:

- 3.1.2 **Time history data**
- 3.3 Data subset referencing in imagery
- 3.5 Color lookup table (palette)
- 3.6 Connectivity list
- 3.7 Spatial feature list
- 3.8 Spatial indexing
- 3.10 SRB and indexing
- 3.11 Database archiving

Unfortunately, the proposed index will not cover use case 3.1.1 (Packet data) because it excludes the ability to dynamically update the index. After some consideration, it was felt that this capability would not be achievable under the constraints of the current project.

The following requirements are based on that use case.

4.1 General requirements

The library should support creation of multiple indices for a given collection.

In the initial implementation, the data types used for key values should be reasonably "standard," to make it easy to access indexes using programming languages other than the implementation language, and to make indexes compatible with other DBMS. Operations In this study, the following operations should be available. Certain other desirable capabilities, which are considered out of scope for this project, are described in Appendix 1.

1. Query an index
2. Create a new index from existing collection of records (filtering)
3. Delete index
4. Cache index, or read index
5. Import index and export index

All but the last of these would be individually callable by applications.

1. Query index

Simple queries should be supported, but the full set of queries is to be determined. The following basic queries should be available.

Access element n. Retrieve the object corresponding to the nth entry in the index.

Selection by criteria. Applications should be able to make simple point and range queries. Possible operations include simple comparison (<, =, etc.) and Boolean (AND, OR, NOT) operations such "Access all elements with temperatures > 50 and latitude=75." Operations should allow both text and numerical queries.

It would be valuable to support user defined query modules (akin to 'stored queries' for a DBMS), and to allow the user to pass in a module to implement queries in its own way.

Range queries on text are complicated and may be avoided in the pass. In principle, we can support queries on arbitrary compound types, but this makes queries more complex and this capability will likely be omitted in the first pass.

A "validate query" operation is also desirable, to confirm that a query is legal against the file and index that it is being applied to.

Other query-related operations of interest include the ability to perform queries across multiple fields, multiple indexes, and even across multiple files.

It is important early on to define a query language, and to provide parsers, checkers, and, if possible, editors for this language, as standalone tools. Import and export to SQL or other languages is also recommended.

2. Create new index from static data

Given a dataset or collection of objects, create a search index for this collection based on

certain criteria. This requires building a query, whose operations are yet to be determined.

To facilitate search queries, it would be good to store the sort criteria with each index, so that it's easy to search given a value ("value" being number, datatype, string, and so forth).

The library should be able to create indexes from a stream of packets, with variable contents per packet.

It is proposed that the initial implementation focus on building indexes to tabular data, and that the resulting index consist of a subset of fields from the original table, plus location fields.

For some applications, a declarative programming model may be preferable to the procedural approach generally used in HDF5. The operation "define an index from schema" would make this possible.

3. Delete index.

Remove the index and update or remove any corresponding metadata from the file.

4. "Cache Index" or "Read Index".

Read the whole index into memory data structures, for optimal speeding up queries and other operations.

5. Import index and export index

It many applications will benefit from being able to import and export indexes. XML is a natural target language for importing and exporting indexes, as it provides a clear path to sharing HDF5 indexes with databases.

4.2 Performance issues

The basic reason for indexes is performance-driven: in a reasonable amount of time, we want to be able to find objects, to iterate through a dataset in a particular order, or to provide alternate views of information. At the same time, for the packet data use case, we need to be able to build an index quickly in real time, as the records stream in.

Because operations on indexes can be performed faster in memory than on disk, a "cache index" or "load index" operation could be very useful when it is possible.

The following treatment describes general performance issues associated with the indexing operations. It will be necessary later to identify rigorous performance criteria for indexing.

Because performance will be important to many applications, the implementation should also provide performance reporting and tuning capabilities.

6. Query index.

Accessing data elements using an index should not take much longer than accessing them directly. Given the limited sophistication of the proposed implementation, queries will need to be simple if reasonable performance is expected. When possible, indexes should

be able to be cached in memory to facilitate this.

For performance, we are worried not just about order of algorithms, but how many disk accesses must be made. For example, queries should only take $O(\log(n))$, but if too many disk reads are involved, performance could still be unacceptable.

Complex queries (i.e. queries that span more than one index) present significant performance challenges. Indeed, optimization of complex queries is one of the primary reasons for using RDBMS. Complex queries will likely be out of scope in this project, but if they are implemented, performance optimization will not be a goal.

7. Create new indexes from static data.

It should be possible to create very large indexes without significant performance degradation. Since computing indexes from scratch can take a long time, care must be taken to make this as fast as possible. It usually requires an $n \log(n)$ sort, and is much slower than searching for two or three elements. Computing indexes from scratch is about as good as sort for iterating through elements once in order.

8. Delete index.

This operation is not expected to present performance problems.

9. "Cache Index" or "Read Index".

This is a performance-*enhancing* operation.

10. Import index and export index

This operation also could affect performance and hence would not be applied in time-critical situations.

Multiple index issues

All of these operations may be required to deal with more than one index, possible large numbers of indices. The result could be serious slowdowns in performance.

4.3 High level or low level library?

Should the index prototype be implemented within the basic library, or should it be a higher level library and API, built using the existing HDF5 data model and library?

Initially, at least, indexes will probably be implemented as high level constructs, with their own API. Since this implementation is to be a prototype, this approach avoids the risk of having to back out significant changes to the library. One concern about this approach is performance. Changes within the library itself, and perhaps even within the HDF5 file format, could certainly achieve better performance. One question is whether the high level approach will be able to achieve adequate performance.

4.4 Data structures

Whenever possible, index data structures should be organized to facilitate operations that are expected. For example, if an index is a collection of fixed-length records, and it is desired to look up items by primary key value, it would be good to require that the index be sorted by its primary key field, enabling a binary search.

Dynamic vs. fixed structures. A critical question in choosing data structures for indexing

is how dynamic they should be? Some advantages and disadvantages of both approaches:

1. Dynamic structures can be altered by making local changes. Fixed structures must be largely re-written in order to be reorganized.
2. Dynamic structures tend to be more complex, and often require extra space overhead. Fixed structures can be relatively simple and do not require extra space overhead to support dynamic changes.
3. Generally, dynamic structures may be more difficult than fixed structure to display and understand by a human viewer. Hence it may be more difficult to adapt viewers to display dynamic structures, and for humans to edit dynamic structures.

The approach to be taken needs to reflect index requirements, particularly operations on indexes.

Building and changing indexes. Most requirements for this implementation do not require dynamic local changes to an index. Only the operation “create local indexes incrementally from streamed data” calls for dynamic changes to indexes. In cases to be covered in the future, large indexes *will* need to be updated, with the cost of re-generating them being prohibitive. Such indexes may be best implemented with dynamic structures. {I’m thinking that initially we don’t need to have dynamic structures, but that the design should include them for the future.}

Querying. Since indexes will primarily be accessed by query operations, it is important to determine the types of queries are best supported. It is not the intention of this project to support all query operations, but rather to target those that are most needed. {Again, we need to look at the use cases. Most likely we initially will want to support binary search, also direct access, and some sequential access. Others, such as n-way search should probably be left for later.}

Multi-layered indexes. It is easy to imagine users creating large numbers of indexes as they analyze their data. The set of indexes itself then could be sufficiently large and complex to benefit from a searchable structure, such as an index of indexes. Multi-layered indexes are beyond the scope of the current work, but may be considered later.

Proposed data structures.

Since a simple table should accommodate all but the “streamed data” case, we propose to use an HDF5 table as at least one initial structure. {I haven’t demonstrated that his this true. Is it?} A table can be described in terms of the HDF5 model as a one-dimensional dataset in which each element is an HDF5 compound datatype, representing an index record. When tables are sorted according to the appropriate search criterion, they can be easily searched to locate records.

For dealing with the “streamed data” case, a more dynamic structure will likely be required. *Tree-structured indexes* involve data structures that enable records to be added and deleted efficiently, while maintaining the ability to look up objects relatively quickly, and to traverse a collection in sequence. *Hash tables* provide fast random access, insertion and deletion, but do not permit fast ordered traversal. The HDF team has recently experimented with *skip lists*, as a data structure that in some circumstances can

provides good search capabilities, and could adapt well to streamed time-sequenced records. At this time, the structure to be used for streamed data is to be determined.

Appendix 1

Some indexing capabilities considered desirable but out of scope for this project

Features to be excluded

For the record, the following indexing capabilities were identified as being valuable for a number of the use cases described in Section 3, but are considered beyond the scope of the current project.

1. Create local indexes incrementally from streamed data
2. Freeze index
3. Re-index (optional)
4. Dynamically update index in background (optional)
5. Validate index

1. Create local indexes from streamed data

Create indexes on the fly as records arrive, but only recently arriving records. For example, as data streams in, it may be desired to build indexes on then most recent 20 records. The result might be a series of small indexes, possibly organized as a ring buffer. These local indexes might be used to build full indexes after all records are collected and ready for archiving.

2. Freeze index.

Mark index as frozen, so that no further changes can be made until the index is “thawed.” “Freeze index” only makes sense when dynamic updating is in use. However, being able to freeze an index and reindex is probably an important use case; applications will often have cycles of adding data and will want to do all indexing calculations when data is not being added.

3. Re-index

Add a batch of new records and/or delete a batch of unwanted index records. This would allow a previously-frozen index to “catch up” without re-creating the original index. This may in the end be no different than #2.

4. Update indexes in background

Keep indexes up-to-date in an environment where the corresponding data is being updated. There are performance limits for this operation, which are discussed in the next section.

5. Validate index

Check that index is up to date and correct. Ideally, this would include integrity constraints, e.g., no pregnant males, etc.

Performance issues

1. Create local indexes from streamed data.

The performance challenge for creating local indexes is to quickly build indexes over a small sub-sequence of records arriving at a high rate of speed. Variables in this operation include (a) the number of fields to be indexed, (b) the number of records constituting a sub-sequence, and (c) the arrival rate of records. Upper bounds for all of these need to be determined. We do know that in one case that records are expected to arrive at a rate between 200 and 500 Mb/sec.

2. Freeze index.

This operation is not expected to present performance problems.

3. Re-index

This could be very time-consuming, but if additions and deletions are small, it may be performed much more quickly than a complete rebuild of the index. One could also possibly keep a list of modified datasets, recomputed indices just for those.

Depending on the algorithm used re-indexing could involve changing the size of indexes, resulting in storage fragmentation. Deletion and regeneration from scratch of indexes could be even worse.

4. Update indexes in background

Keeping indexes up-to-date in an environment where then data is being updated takes resources away from the update operations, and hence is likely to slow down the update operations. Dynamic updates of indexes could also pay a high synchronization cost. It also constrains the algorithms that can be used. If it can be done, however, it should require a smaller overall performance hit than freezing indexes and regenerating them, especially for large indexes.

5. Validate index

This operation needs to be fully specified and analyzed, but seems likely to be one that could affect performance and hence would not be applied in time-critical situations.