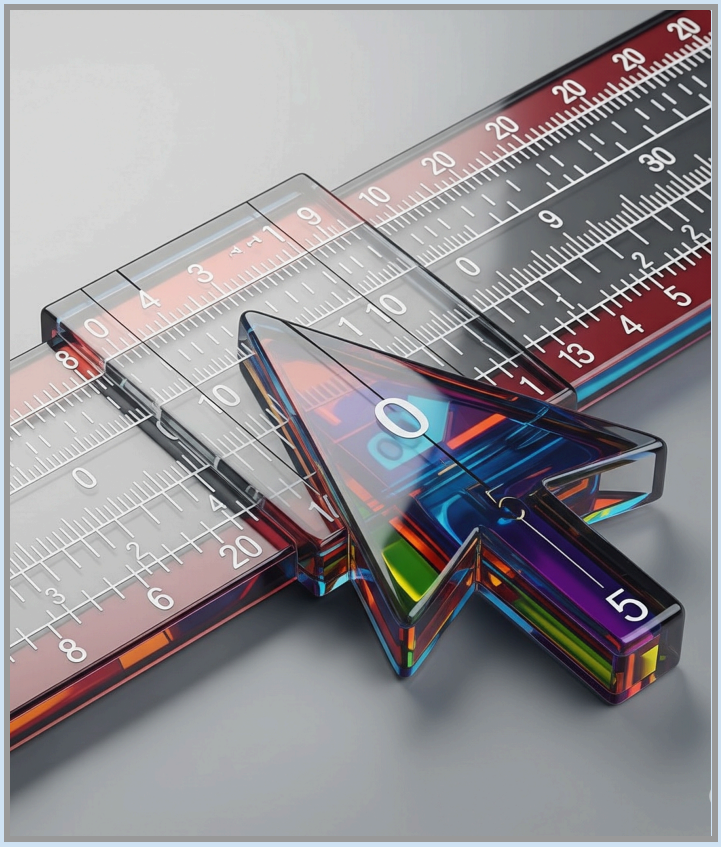
METASPEX

Metaspex Technical Insights

V - Indexes, Cursors, Multitenancy & Paginated Services



Introduction

Acquiring a document from a database can be done in different ways. So far we've seen that merely following a **link** is enough to instruct Metaspex to retrieve a document. This does not require any database API or query language. No explicit deserialization is involved.

Another possibility is to use the static **get** function generated for every **root**. It takes a database connector, with the database-specific details specified in the configuration file, and the document identifier. It returns the corresponding document if it can be found in the database. Here also, from a specification standpoint, it's just a function call, that's all.

In both cases, the mechanics involved behind the scenes are very fast, as underlying document databases implement document retrieval from their document identifier in a very fast and scalable manner.

But something is still missing. Sometimes it is necessary to find one or several documents based on their characteristics, not just from their identifier. By that we mean that sometimes indexing documents based on one or several of the **slot**s, or the **link** relationships they bear, can become necessary. Take a use case like "find me all the people with a last name and first name between these two values." This cannot be achieved with the two former techniques.

Additionally, documents contain **own** relationships, and the **slot**s and **link**s we want to use for indexing can be located deep inside the document, being only accessible from the **root** object through a path of one or several **own** relationships.

This is where indexes and **cursor**s come in handy.

We'll also address in this document how Metaspex implements multitenancy automatically and almost transparently, and we'll discuss paginated services, which are specifically dedicated to automating concisely powerful web services which allow clients to browse through documents with a "window" running on an index.

Index definition and creation

When it comes to indexes, Metaspex follows the philosophy applied consistently across Metaspex: making Metaspex source specification independent from underlying database product choice details. In fact, indexes are not even defined in the source code. Instead,

they are defined in the application configuration file. Their usage in the specification is identical across database products.

The creation of the indexes is performed through a simple call to a Metaspex utility (**hx2a configure**), which reads the configuration file, creates the databases listed there if necessary, and performs index creation in the respective databases automatically. It adjusts to whichever database product has been selected, to invoke the right commands.

We have a dedicated technical insights document on the configuration file, encompassing the database definitions. In this document we're going to focus only on the index definitions.

With Metaspex ontologies you can create indexes over data on the **root** object, but also on data on objects deep in the document attached to the **root** object. This means that the index definition can refer to a path to that piece of information. That path uses the **tag**s chosen in the ontology (**tag**s for **own** relationships of all kinds, and **slot**s). You cannot cross a **link** for indexing, meaning that you cannot include data elements from linked-to documents in an index. (You can, however, index on the unique identifier of the linked-to document.)

As databases can create an index upon several pieces of data, the definition of an index incorporates a vector of **tag** paths:

```
index "directory" "hx2a_org__cty" "@o" "@c$:a.ctry:a.cty"
```

This index definition comes with Metaspex in file "dirindex.conf" alongside many other index declarations. It is an index which is automatically created when using the directory. The directory is where **users**, **organization**s and other useful security-related documents live.

The first string after the keyword "index" is the name of the database ("directory"); the second is the name of the index ("hx2a_org__cty"), as it will be referred to in the specification; and the third is the **tag** of the **root** type (the user-defined second argument of the **HX2A_ROOT** macro), "@o". Here, it is the **tag** of the Metaspex predefined **organization** type.

Last we see the string containing a vector of **tag** paths. The three paths are separated by colons, while the paths themselves use a period ("@c$**:**a**.**ctry**:**a**.**cty"):

- "@c$": the **link** to the community the **organization** belongs to (we'll see in the document pertaining to security what a community is);
- "a.ctry": the path to the country **slot** in the **organization**'s address. The **organization** bears an **own** relationship to the **address** type, which itself contains a country **slot**.

The **tag** of the **own** relationship is “a”, while the **tag** of the country **slot** in the **address** type is “ctry”;
- “a.cty”: the path to the city **slot** in the **organization**’s **address**.

Should we have a **root** type with **tag** "person", bearing **slot**s with **tag**s "family_name" and "given_name", and an **own** relationship to an **address** with **tag** "addr", if we wanted to declare an index "my_index" in database "hx2a" over the family name, then given name and then address’s country, we would declare it like so:

```
index "hx2a" "my_index" "person" "family_name:given_name:addr.ctry"
```

The index created in the database will be lexicographically sorted, like a phone book. In this example, it will contain something like the following entries:

```
"Miller", "Ruth", 840
"Miller", "Steven", 840
"Smith", "John", 826
"Smith", "John", 840
"Smith", "Samantha", 250
"Smith", "Samantha", 840
```

Metaspex allows us to index over:

- **link**s, adding the suffix “$” to the path;
- **own_list**s and **own_vector**s, adding the suffix “*” to the path. In this case, each element within the composite relationship is enumerated and the path applies to every element found.

The two can be combined, by adding “$*” to the path, like in this predefined directory index on licenses:

```
index "directory" "hx2a_lic__custn" "@l" "@c$:C:S:w$*:s"
```

Descending indexes are specified using the symbol “-” at the end of a path:

```
index "hx2a" "mydescidx" "mytype" "C-:S"
```

The index will be descending on the first part of the key. (This feature is not available with CouchDB.)

Browsing an index with a cursor

In some use cases, you may want to apply some kind of operation (read or write) to a set of documents in an index. This might involve all the documents in the index, or only some of them.

If only some of them are to be scanned, Metaspex will require an interval of keys. This interval can be open on one end, or closed on both ends.

The number of documents involved might be significant, so it might be preferable to process a specific limited number of documents at a time, while iterating on the index using something like a window always containing only that limited number of documents, until the end of the index or of the key interval boundary is encountered.

This is the purpose of Metaspex **cursor**s.

Metaspex **cursor**s are strongly typed, meaning that they will return a number of documents matching the selection criteria, and these documents will be accessible to the specification as fully deserialized typed objects.

There are three behaviors offered by Metaspex **cursor**s:

- **cursor**s staying on one key value, which may be partially specified;
- **cursor**s starting at a given key value set and which keep going until the index has been completely scanned;
- **cursor**s starting at a given key value set and stopping after a different key value set has been encountered, or the index has been completely scanned.

When a set of keys is supplied, documents matching these keys will be included in the index. This means that **cursor**s manage closed key intervals. Documents will be sorted lexicographically, as they are in the database index.

The functions provided to create these three behaviors are:

- **cursor_on_key**<DocumentType>;
- **cursor_from_key**<DocumentType>;
- **cursor_on_key_range**<DocumentType>.

They all return the Metaspex type **cursor**. That **cursor** will behave according to the function used to create it.

In the example above, if a **cursor_on_key_range** is selected and "Miller", "Steven", 840 is specified as the lower bound of the interval, while "Smith", "Samantha", 250 is the upper bound, the **cursor** will return only the following entries in the index:

```
"Miller", "Steven", 840
"Smith", "John", 826
"Smith", "John", 840
"Smith", "Samantha", 250
```

The bounds given to the **cursor** do not need to exist in the index; they just define an interval.

Note that this is different from selecting an interval of family names and an interval of given names and an interval of country codes. Doing that efficiently when there are lots of entries in an index is done using a multidimensional index, not a **cursor** on an index. Multidimensional indexes are covered in a separate document.

No SQL or query language is needed to operate **cursor**s. They operate transparently and interact with the databases you choose in the configuration file using the appropriate database APIs, hidden from view. You can deploy the same binary application on different database products just by changing the configuration file that Metaspex will read when the application is started.

Cursors guarantee that they perform a bare minimum of searches in the index. Technically their complexity is O(log(N)), N being the number of documents in the index.

If you need to perform more complex operations, like filtering, or if you need to use set theory (union, intersection, etc.), you have the C++ Standard Library, which manages all sorts of data structures, at your disposal. The perfect usage of indexes and the complexity of the use case you specify is completely under your control, as C++ algorithms have guaranteed complexity. Fighting with a DBMS and its execution plans becomes unnecessary.

When one of the three **cursor** creation functions is used, you must supply some arguments. All of them take a limit, which is the number of documents acquired from the database at every iteration of the **cursor**.

- **cursor_on_key** takes the key value to stay on;
- **cursor_from_key** takes the key value to start from;
- **cursor_on_key_range** takes the start and the end key values of the interval.

Here is a snippet of code showing how to use these functions on the family name, given name and country example:

```
#include "hx2a/cursor.hpp"
#include "hx2a/cursor_on_key.hpp"
#include "hx2a/db_connector.hpp"

#include "person.hpp"

using namespace hx2a;

void f(){
   db::connector cn("my_db");
   cursor c = cursor_on_key<person>(cn->get_index("my_index"), {.key = {"Smith",
"John"}, .limit = 128});
   // Iterate over the cursor to scan each person from the key supplied.
}
```

We supplied only two values for the key, instead of the three expected by the index. Metaspex will complete the values in the key automatically. The **cursor** will retrieve 128 documents at each iteration (128 is the limit), until it has exhausted the index completely. On the index above, it will return:

```
"Smith", "John", 826
"Smith", "John", 840
```

And then it will stop.

If the key given to the **cursor** was simply {"Smith"}, the documents returned would be:

```
"Smith", "John", 826
"Smith", "John", 840
"Smith", "Samantha", 250
"Smith", "Samantha", 840
```

If the **cursor** was initialized using:

```
cursor c = cursor_from_key<person>(cn->get_index("my_index"), {.key = {"Smith",
"John"}, .limit = 128});
```

The iterations would stop only at the end of the index, returning:

```
"Smith", "John", 826
"Smith", "John", 840
"Smith", "Samantha", 250
"Smith", "Samantha", 840
```

And if the **cursor** was initialized like so:

```
cursor c = cursor_on_key_range<person>(cn->get_index("my_index"), {.lower_bound
= {"Smith", "John", 840}, .upper_bound = {"Smith", "Samantha", 250}, .limit =
128});
```

Metaspex does not need to complete the values forming the key, as we specified three values. It would return:

```
"Smith", "John", 840
"Smith", "Samantha", 250
```

Please refer to the reference manual for details about how to use these arguments, and information on additional ones. Note that the sets of keys do not have to be complete. Metaspex will automatically complete the sets of keys with the proper minimal and maximal values. This is very handy when you want to reuse an index, as indexes are fairly costly in a database, and for a number of new use cases you need additional keys in the index. You can add them in the configuration file, recreate the index, and existing code not using the additional keys will not be broken–it will keep working as expected.

A typical loop processing the documents scanned by **cursor** c looks like this:

```
while (c.read_next()){
   // rs contains a number of documents corresponding to the specified limit.
   auto& rs = c.get_rows();

   for (const auto& r: rs){
      // Do something with document r.get_doc() which returns an rfr.
   }
 }
```

for_each_row, for_each_doc

If you prefer a specification style with a single loop, you can use the **for_each_row** or **for_each_doc** abstractions which do exactly that. They take a lambda expression, and the limit is used only behind the scenes.

A typical code using **for_each_row** is:

```
for_each_row(c, [](const auto& r){
   // Do something with document r.get_doc() which returns an rfr.
});
```

This function will iterate upon all documents encountered by the **cursor**.

As indexing is possible on **own_list**s or **own_vector**s, the same document might be present several times in the index. In case you want to perform the operation only once on each document, you can use the slightly more costly **for_each_doc**:

```
for_each_doc(c, [](const auto& r){
   // Do something with document r.get_doc() which returns an rfr.
});
```

Multitenancy

Multitenancy is the ability to separate for some processes the operations on different tenants' data. Multitenancy is not a protection against malicious developers, it is a protection against bugs and especially against application users invoking web services on data to which they should not have access.

Metaspex has an extensive identity management framework which incorporates the concept of **organization**. It is not the object of this document to elaborate on that. Suffice to say that users can be affiliated with one or multiple **organization**s, and **organization**s are organized in tree-like structures, where at the top of these structures are special **organization**s called "communities."

Multitenancy requires using Metaspex's built-in services for a user to log in and to select their current **organization**.

Metaspex multitenancy automatically creates a **link** between the documents created by a tenant, meaning a **user** operating on behalf of an **organization**, and the community the selected **organization** belongs to.

This **link** relationship is a Metaspex **strong link**, meaning that if the community is removed from the directory, all data belonging to that community will be removed automatically by Metaspex referential integrity.

In order to declare a type as being subject to multitenancy, it is only necessary to replace the base type **root**<> with the base type **entity**<>. The **entity** type is obtained by including the header file "hx2a/components/entity.hpp".

The macro HX2A_ENTITY is available. Conveniently, it is invoked identically to the macro HX2A_ROOT.

When declaring an index, you can specify that it is a multitenant index by simply adding the “@c$” key first. “@c” is the **tag** of the **link** pointing at the **entity**’s community **organization**. As we saw, to **index** over a **link** is just a matter of adding the “$” suffix, hence “@c$”.

When creating a **cursor** on a multitenant index, the leading key to be supplied is the document identifier of the **organization** given by Metaspex to the web service. When using the appropriate web service prologue (see reference documentation), the community the user selected after a login is automatically passed to the web service.

As a result, **cursors** will browse through the multitenant index scanning exclusively the documents belonging to the user’s community: in other words, the documents belonging to their community’s tenancy. The **cursor** cannot cross the tenancy barrier.

You can still specify processes which can cross the barrier using non-multitenant indexes and the **cursor**s associated with them. This can be useful when some users have specific privileged roles, to mention just one example.

Paginated services

The need to browse through an **index** is not limited to implementing some use cases inside a backend. It occurs very frequently at web services level. Business application user interfaces are full of tabular lists with a scrollbar or “next” and “previous” buttons.

Metaspex calls these paginated services. When engineered by hand, they can become a nightmare for the developer. They are very complex, as they involve stateful interactions

between the client and the server. Given that they are everywhere in business applications, the pain they create is significant.

This type of situation illustrates the power of Metaspex as a high-level specification language for business applications. It provides a ready-made, concise, sophisticated and efficient abstraction to automate all paginated services entirely.

Every time the **paginated_services** abstraction is used, two web services are defined automatically, the “next” and “previous” web services, allowing clients to navigate on the index. This set of two web services is associated with one user-specified index.

The **paginated_services** abstraction is a C++ template taking at least three template parameters. Additional parameters are optional. Here are the three mandatory parameters:

1. A string, which will prefix the “next” and the “previous” web services. The suffixes "_next" and "_prev" will be added to form the names of the two web services generated;
2. The **root** type of the documents indexed (MySource in the example below);
3. A projection functor turning each document into “row” payloads (of type MySourceRow below).

The next and previous services will return multiple rows, which will be the result of calling the projection functor applied to each document encountered during the scan of the index.

A typical way to use **paginated_services** is shown below.

```
#include "hx2a/conf.hpp"
#include "hx2a/paginated_services.hpp"

// Declaration of MySource.
#include "mysource.hpp"

// Some declaration of MySourceRow, which contains the relevant details from
// MySource to be returned as rows when a pagination service is called.
// ...

using namespace hx2a;

rfr<MySourceRow> compute_mysource_row(const rfr<MySource>& source){
   // Some code creating a MySourceRow with make from the source reference,
   // and returning it.
   // ...
}

// Declaration of a singleton of type paginated_services. It will create the
// next and previous web services.
paginated_services<"my_paginated_service", MySource, compute_mysource_row>
   _my_paginated_services(config::get_id("my_database"), "my_index");
```

We can see that the string "my_paginated_service" was supplied as the first argument of the **paginated_services** template. It means that the two web services automatically generated will be named my_paginated_service_next and my_paginated_service_prev.

The type of document present in the index is MySource, which is the second argument.

The projection functor capable of turning an **rfr** to a MySource document into an **rfr** of type MySourceRow is compute_mysource_row.

The declaration above creates a global variable _my_paginated_services, to which is assigned a singleton of type **paginated_services<>**. It is not necessary to refer to this global variable anywhere else, but the declaration serves to define the “next” and “prev” services.

To be built, the **paginated_services** requires arguments. They are supplied as:

- **config::get_id**("my_database"), and
- "my_index".

The first call accesses the database identifier (a UUID) declared in the configuration file, which corresponds to the configuration file database name "my_database".

The second argument, "my_index", is the name assigned in the configuration file, for the index the paginated services will scan. This index must index documents of type MySource.

The compute_mysource_row projection functor is completely user-defined. It is called automatically by Metaspex, and Metaspex will call it with an **rfr** to a document it encountered when scanning the index. The functor can access anything it needs in the MySource document. It can access **slots**, it can cross **own** relationships, and it can even cross **links** if necessary.

For simplicity, we're not showing all the parameters that the **paginated_services** template accepts. Please refer to the reference guide for that. We can say a few words, though, about what the optional parameters can do:

- A **prologue** can make security checks, for instance, to verify that the user calling the next and previous web services is authorized to do so;
- A user-defined type can be added to the query payloads accepted by the paginated services, so that additional data coming from the client can be passed along to the functors invoked by the paginated services;
- A **key massager** functor can be supplied to inject leading keys into the index keys sent by the client, to ensure that the paginated services scan only a subset of the index. That way the paginated services will navigate only among documents bearing the same leading keys. This supplies a level of isolation for the navigation. This allows, for instance, for the implementation of multitenancy.

The JSON payloads offered to the client side to call paginated services are pretty sophisticated. They can contain the following keys and corresponding values:

- "limit": its value is some JSON number specifying the maximum number of documents to be returned. When not specified, it defaults to 10. A hard internal maximum limit is set for this value (typically 100). It is accepted by the two services.
- "lowerkey": its value is a JSON array specifying the key at which the "prev" service calls will reach an end. It is to be given only to the first service call.
- "startkey": its value is a JSON array specifying the key at which the first document returned must start. It is to be given only to the first service call.
- "upperkey": its value is a JSON array specifying the key at which the "next" service calls will reach an end. It is to be given only to the first service call.

The next and previous web service calls return a **cursor** JSON key, which just needs to be echoed back at the next call to navigate forwards or backwards.

As a result, the JSON reply will look like:

```
{"cursor":{...}, "rows":[{"id":"...", "k":..., "r":{...}}, {"id":"...", "k":...,
"r":{...}}, ..., {"id":"...", "k":..., "r":{...}}]}
```

Each element of the JSON array "rows" will contain the document identifier of the source document, while the keys found in the index and the "r" will contain the serialization of the projection.

Conclusion

With **cursor**s, we have completed our survey of the various ways to acquire documents from databases. As with other Metaspex features, **cursor**s eliminate the need to use database-vendor-specific APIs and query languages. They elevate code as a high-level abstraction devoid of implementation details and on par with a technical specification. They offer performance guarantees, by ensuring that no full collection scans ever take place.

On top of **cursor**s, Metaspex offers automatic multitenancy at minimal specification cost, making applications safer while allowing economies of scale when customers share the same hardware infrastructure, networks, processors, RAM, and disks.

Also on top of the index, powerful paginated web services, which are tricky to build by hand, automate entirely and safely the engineering of the widespread requirements of windowed browsing through documents based on complex indexing and projections.