NutleyDB Spec

Joseph Victor

October 28, 2013

Contents

1 Overview

NutleyDB is a practical implementation of David Spivak's Simplicial Database model [**?**]. It is a distributed column-oriented DBMS where each database is immutable and data can be shared across databases. Roughly speaking, a simplicial database is a specified by a simplicial set X, the schema, with one "column" placed on each vertex and a "table" on each higher simplex, the sheaf of records. We will use the immutability of the data to allow unions and more general colimits of simplicial databases, both to organize column segmenting and data distribution. We will find that this model is extremely flexible and allows for considerable consolidation of ideas.

1.1 Simplicial Databases

Let $\mathcal{T}$ be a small concrete category whose objects are data types you want to be able to store and whose morphisms are some set of functions between those datatypes, for instance the category of serializable Haskell types and Haskell functions. Let X be a simplicial set (in practice a simplicial complex) with each vertex x labeled with an object $a \in \mathcal{T}$, denoted $\mathcal{U}_X(x)$. For a simplex σ, let

$$\mathcal{U}_X(\sigma) = \prod_{x \text{ a vertex of } \sigma} \mathcal{U}_X(x)$$

We call such a labeled simplicial set a $\mathcal{T}$-schema, or just a schema when $\mathcal{T}$ is understood. The projections $\mathcal{U}_X(\sigma) \to \mathcal{U}_X(d_i\sigma)$ form the base of a sheaf on X, called the universal sheaf of records on X. The sections over a union of simplexes $X' \subset X$ is specified by one element of $\mathcal{U}_X(x)$ for each vertex $x \in X'$.

Let X be a schema and let K be a sheaf of sets on X such that $K(X')$ with a map $\tau : K(X') \to \mathcal{U}_X(X')$. Together the pair K, τ forms a sheaf of records over X, which we'll sometimes call an instance of the schema X. In some sense, K is just a nuisance put in to organize duplicate rows and make sure the restrictions are just projections. What is actually specified is, for each simplex σ, a multiset of elements from $\mathcal{U}_X(\sigma)$ (the image of $\tau(K(\sigma)) \subset \mathcal{U}_X(\sigma)$ counted with multiplicity) and maps from the multiset on σ to the multiset on each face $d_i\sigma$, such that the obvious diagrams commute. In essence, you specify some records in each $\mathcal{U}_X(\sigma)$ and the projection maps onto each face.

These ideas unify a handful of ideas in normal databases. First of all, data in one column can be used in multiple tables by building simplexes which share vertexes, or more generally two different tables can share the same projection if they have the same face. This idea is analogous to foreign keys in traditional SQL. Additionally, looking at the sheaf over a union of simplexes gives an equi-join of those simplexes over their intersections, so these joints live implicitly in your database, and the sheaf laws encode possible query optimizations (like pushing projections under joins).

Finally, the notion of nullable values in SQL tables (which have a reputation for being confusing because null≠null), can be encoded as records on a face not in the image of the restriction map. This satisfies the intuitive use of nulls, that they are "the stuff that I could put in a projection of a table that don't quite fit in a table". Indeed, encoding nulls in this way removes the distinction between inner and outer joins, since if σ and τ are simplexes then the records in $K(\sigma \cup \tau)$ would be analogous to the SQL inner join, but the codomain restriction map to σ and τ contains the unmatched entries, which in this scheme is analogous to the unmatched entries being matched with nulls. The joins implicit to a given simplicial database therefor "have it both ways", and can be interpreted as inner joins, left outer joins, right outer joins, full outer joins or even cross joins if the simplexes in question are disjoint; any SQL join over equality can be encoded in this way.

1.2 Column-wise Data Storage

Column-oriented DBMSs (columnar-stores) are gaining popularity do to their ability to laugh in the face of the time-memory tradeoff. Columnar stores tend to use less space than tuple-stores because data from a single column is more homogeneous, so can be better compressed. They also can preform better on certain OLAP queries, since such queries often only use a few columns at a time. This can be a win on several front, since less data needs to be touched, the data touched will be closer together, and operations can be done in cache-sized batches which run very fast on modern computers. The downside of course is that transactions (small inserts, updates and deletes) can be slow since it involves touching potentially all the rows and possibly requires compression and decompression.

The simplicial database model naturally lends itself to column-wise data storage. Simply put a column on each vertex. On each higher simplex, you essentially need to specify for each record a tuple of things in the column. This can be done mostly implicitly, which is good, since there are 2^n sub-simplexes of an n-simplex!

This is clearly more flexible than a traditional column store, in which each column has the same number of entries and the records are just "horizontal slices" of the columns. While such a design is easy to build as a simplicial database instance, you can follow other patterns as well. For instance, we remarked above about how to store nullable entries in this scheme. You can also sort your data (for better compression) or if you have only a few unique values in a column you could only store them once and have the higher simplexes figure out which records go with which, which is equivalent to the standard idiom of building an auxiliary table with a foreign key, except that it can be done at the level of any simplex and "storage technicalities" do not show up in the database schema.

1.3 Database Aggregation

Database instances is NutleyDB are *immutable*. This is contrary to conventional wisdom, and will make certain queries (mostly OLTP queries) impossible to do efficiently. However, it will allow for many simplifications and benefits which we hope will outweigh missing features. One of them is simple share-nothing database aggregation. Let X be a schema and $K_1, ..., K_n$ be some database instances over X. Then we can form the sheaf $K = \bigcup_i K_i$ over X. This need not be done explicitly; K merely needs references to the K_i, and queries on K can be translated to queries on the K_i. If the K_i are distributed across many machines, K is like an aggregator in a traditional distributed database. Each K_i could be local or remote, and can be queried directly by another client or aggregated by another aggregator without causing problems. Furthermore, if the aggregator decided that it would be a good idea to explicitly form $K_2 \cup K_5$, move data between remote machines or perform some other compactization or optimization, it can, so long as it does not destruct its parts.

This unifies two important ideas. In a column oriented database, the entire column is seldom stored together. Instead, it is chopped up into segments, possibly based on which rows were inserted near each other temporally or by some other criteria. These segments are often stored in a compressed format and our typically immutable or mostly immutable. Since a given arrangement of segments could be suboptimal, the database engine will try to find opportunities to merge segments, remove deleted rows in batch, sort things or move them around in some other way. The entire column then is the union of all the segments, which would probably be kept locally, and the above formalism describes this phenomenon well, where each segment is itself a database instance. On the other hand, a distributed database could work by having aggregator and leaf nodes, where the leaves correspond to remote instances of the database and queries on the aggregator are converted to queries on the segments. These two similar ideas become unified in this model.

1.4 Simplicial Challenges

The simplicial database model is extremely flexible. In the most complicated case, you can a table with n columns can have specified $O(2^n)$ record sheaves and $O(2^{2n})$ restriction maps. We will partially remedy this by requiring very complicated databases to have very complicated specifications; the storage requirements should be polynomial in the input as a string.

2 JoSQL Queries

We will list a small set of underlying queries which all reasonable queries can be compiled into. This language is designed to be a small, easily implemented subset of NutleyDB's full query language upon which we shall bootstrap.

2.1 Create Database

```
database <dbname> where
   <vertexname> :: <vertextype>
   ...
   <simplexname> = (<vertexname>,...,<vertexname>)
   ...
   <simplex> = <simplex>
   ...
```

where

```
<simplex> := <vertexname> | <simplexname> | D <int> <simplex>
```

This creates the scheme with the vertexes given and simplexes given, and specified relations between simplexes. Creating a k-simplex implicitly creates the intermediate simplexes. Notice that nothing stops a user from creating a The relationships specified at the end say how to "glue" the top dimensional guys together. An exception is thrown if the gluing is over simplexes with type disagreements, including different dimensional simplexes.

2.2 Create Instance and Load Data

Definition 2.1 (Simple Record)**.** *Let $X' \subset X$ be a simplicial subset of X. Then the sheaf of records over X' can be specified directly. Let $x_1, ..., x_n$ be the vertexes of X', and let S be some nonempty subset of the x_i. Then a simple record over X' is an element of the sheaf over simplicial subset of X' spanned by the x_i. Said another way, it is a record in the table with columns from S. A set of simple records R defines a sheaf $\mathcal{K}$ as follows. Write R_S as the set of records defined with the vertex subset S. If σ is a simplex with vertex set T, then there is a natural projection from R_S to $\mathcal{U}_X(T)$.*

$$\mathcal{K}(\sigma) = \bigcup_{S \supset T} proj_T^S(R_S)$$

The restriction maps are the evident ones.

Definition 2.2 (Record Pullback)**.** *This is an obvious but hard to say notion. Basically, if all the faces of σ have a sheaf on them then you can put a sheaf on σ by filtering the sheaf over the union of the faces in some way. The filtering function f assigns to each record r over the union of the faces of σ a number (possibly zero) of copies of r in the sheaf over σ, each restricting to r. Of course, the function is aloud make arbitrary choices depending on the order it considers the records on the faces, so the function is pretty general, so the type of f is $\mathcal{M}(\mathcal{U}_X(\sigma)) \times \mathcal{U}_X(\sigma) \to \mathbb{N}$.*

To instantiate a simplicial database over a schema X, one must define the sheaf of records on each simplex and the maps. There are two ways to do this. One is to specify a set of simple records on a set of simplicial subsets. The other way is to "pull back" the sheaf on the face of a simplex to the sheaf on that simplex.

To instantiate a database

```
instantiate <dbname> with
   <instantiation>
   ...


<instantiation> := simple <subset> by [<simple record>]
                 | <pullback> <simplex> by <pullback function>
```

any function of returning simple records of the correct type is sufficient for defining these simple records (for instance, a selection). The order of the instantiations matter. Each instantiation unions in new records to the last, and the pull back depend only on previously defined records. The order in which the pullback considers records is implementation defined but consistent.

One important way of defining sets of simple records is, naturally, by loading a csv file.

This is the only query that writes to disk.

Pushforward and Inverse Image

We got em. They are "implicit".

Data "mapping" is a special case of pushforward

Restriction is a special case of Inverse Image where the simplicial set map is an inclusion and the type maps are identity.

The List Functor, and MonadPlus Functors

For each vertex $x \in X$ takes $\mathcal{U}_X(x)$ to $[\mathcal{U}_X(x)]$. We call the new scheme $[X]$ Replaces each section with a singleton section containing the list of everything from that section. Restrictions are maps of restrictions. This is an "in memory" construction, hopefully making good use of laziness. The new sheaf is called $[\mathcal{K}]$.

A pushforward of this guy can be an aggregates If $X = *$ where $*$ has type `Int`, the map `sum :: [Int] -> I` gives a map of schemas $[X] \to X$, and the pushforward is gives the aggregate.

This construction works with any MonadPlus.