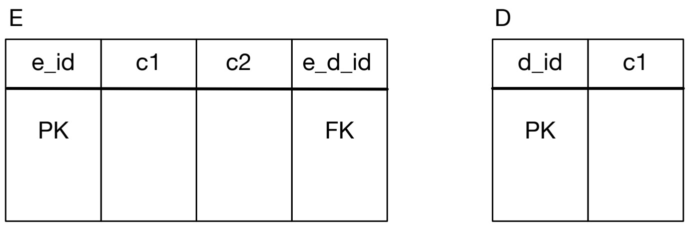
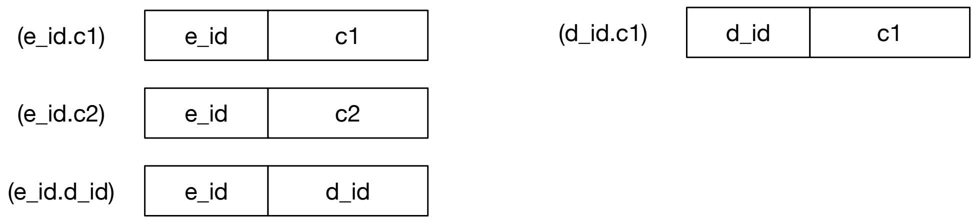
1. Low Level Query Translation

In this document, we will present different SQL queries and their equivalent in L2 which would constitute the low level part of our naive NoSQL compiler. The low-level part of the compiler focuses on transforming single queries into L2 (in this case Riak and Antidote) facilities, which are usually executed within a sharded datacenter (as oppesed to system-wide distributed transactions) Translating the input SQL programs (that wrap low-level queries of this document) will be presented in the next writeup.
In the following examples we will assume an input SQL data model:



We also assume that the output of the queries is always stored in the variable v. We also assume full isolation in cases when a node performs multiple L2 operations (i.e. there is no interference between operations of different processes within a single node).
We start the translation by presenting the initial L2 objects (we will incrementally update this model given new queries to be translated):



1.1. single predicate, equality search

1. On the primary key:

SQL: `v = SELECT c1 FROM E WHERE (e_id = key)`

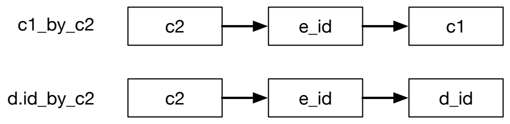
L2: `v = (e_id_c1).kvREAD(e_id=key)`

SQL: `v = SELECT * FROM E WHERE (e_id = key)`

L2:
```
v.fst = (e_id.c1).kvREAD(e_id=key);
v.snd = (e_id.c2).kvREAD(e_id=key);
v.thd = (e_id.d_id).kvREAD(e_id=key)
```

2. On a non-primary key (e.g. c2): In this case we have two options: *a*) Using a denormalized datamodel which avoids datacenter-wide scans but adds extra anomalies due to the unsync copies of entries. *b*) Using Riak's map-reduce capabilities or Secondary Indexes, both of which are easier to implement and maintain but require data-center wide scans with growing complixty proportional to the number of partitions and the number of entries. Neither of them are recommended for production usage.

(a) Denormalization: We should update the L2 data-model by *adding* two new buckets: *c1_by_c2* and *d.id_by_c2*, each keeping a map from c2 column type to a map from e_id to c1 and d.id respectively:



SQL: `v = SELECT * FROM E WHERE (c2 = key)`

L2:
```
v1 = (e_id.c1).mapREAD(e_id=key);
v2 = (e_id.d_id).mapREAD(e_id=key)
Let v = zip(v1,v2)
```

(b) MapReduce or Secondary Indexing: Basho documentations state:

"We recommend running MapReduce operations in a controlled, rate-limited fashion and never for realtime querying purposes."

and

> “ If your ring size exceeds 512 partitions, secondary indexes can cause performance issues in large clusters.”

Which forces us to choose the former approach when designing our compiler. However, in order to make this document complete we will briefly discuss them here. Riak supports simple secondary indexes on objects by tagging them with string or integer values. The tags indexes are maintained at the same shard where the data resides (which explains the performance issues when growing the cluster).
In this example, without denormalizing the data model, we can tag (e_id.c1) and (e_id.d_id) objects with values of c2 (let's assume c2 is an integer column):

L2:
```
(keys,v1) = (e_id.c1).get_index('c2', key);
Let v2 = map(k=>(e_id.d_id).kvREAD(e_id=k)) keys
Let v = zip(v1,v2)
```

As we can see, this approach is not easier than the denormalized version. We could have made this case a little simpler by changing our initial fine-grained model into a model where all columns reside in the same Riak object (coarse grained modeling), which would have eliminated the need for the iterative reads, but even in that case Riak seems to update the tag table and the entries in a weakly isolated fashion which might cause some anomalies[1].

1.2. Multiple predicates, equality search

Since Riak does not support partial key queries, in this case, we should follow a hybrid approach where we make use of both denormalization[2] (if the PK is not used in the predicates) and a MapReduce function (for filtering out the unwanted entries according to the rest of the predicates[3]).

[1]This claim requires experiments, I could not find anything in the documentations

[2]Note that, we could have simply used MapReduce functions without denormalization and allowed DC-wide scans, however the Basho documents state: "Do NOT use MR when you want to query data over an entire bucket. MapReduce uses a list of keys, which can place a lot of demand on the cluster."

[3]Or fetch all entries satisfying one predicate and trim them locally

For example, if we wanted to query all entries of the table E where (`c2` = `key1`$\wedge$ `e_d_id` = `key2`), we could have denormalized the data model and queried riak similar to 1.1.2.(a) with a map function to filter out the undesired entries:

```
function(value, keyData, arg){
  var data = Riak.mapValuesJson(value)[0];
  if (data.e_d_id == key2)
    return [ data ];
}
```

1.3. Inequality Search (Single/Multiple Predicates)

In this case, denormalization does not necessarily eliminate the datacenter-wide scans, since the range of keys queried might reside on multiple (or even all of the) shards. Note that using MapReduce functions we can collect the entries satisfying all predicates from the shards in the DC, however, there is no guarantee that the entries from different shards would be from the same version.
For example, consider a datacenter with three shards. After S1 queries S2 and S3 to collect a set of entries, and after S2 computes its response, a new remote range update might arrive at the datacenter. Now S3's result is going to contain the newly arrived update but S2's result lacks it. This anomaly which would cause an inconsistent state read on range queries, does not exist in unsharded databases (and trivially in relational databases).

Riak and other well known NoSQL stores do not provide any facitility that could be used to fix this problem (I guess since the connections within a DC are usually very fast, this problem is not prevalent in production. However since we are bulding a verified implementation we should handle and eliminate this anomaly as well).
In order to avoid this anomaly, we can use higher level facilities usually offered by extension tools running on kv-stores. For example we can use Antidote's read-only (write-only) transactions to guarantee that all such queries (range updates) end up accessing (updating) the database instantaneously.
In the following example, I will sketch the translation of a simple range query. The high-level program translation which I will present in the next write-up will include exact notions of WO and RO transactions of Antidote and our general purpose compilation scheme.

Example: Assume a simple case where we want to query c1 field of all the entries from E with ids less than 100:

- **SQL:** `v = SELECT c2 FROM E WHERE (c1 < 100)`

E

e_id	c1	c2	e_d_id
1	{x}		
12	...		
2	{z,y}	...	...
111	...		
3	{}		

- **L2**:

```
\\ first create a list of object/keys to be read
for i in [0..100]:
        Obj[i] = {i, antidote_crdt_set,e_id_c1}

\\ read values from the given key-list in a transaction
{ok, Result, CT2} =
        rpc:call(Node, antidote, read_objects,
        [CT1, [],Obj]).

\\Result[1]= {x}
\\Result[2]= {z,y}
\\Result[3]= {}
\\ ...
```