Database Setup for DC3+

For DC2, each nightly pipeline run was confined to reading and writing data from a private database - pipeline initialization included a step which created the tables for the DC2 schema and copied in the input Object catalog (consisting of 417,327 objects). If subsequent data challenges have large enough input Object catalogs, then our development cluster won't be able to support scores of run databases (as in DC2) unless we come up with some way to economize on disk space.

This page presents three methods for reducing the cost of per-run databases and a proposal for more aggressively deleting database outputs of a run.

Particularly important is that a solution should not inhibit cross-run data comparisons. In relation to this point, a desire to have consistent entity ids across nightly pipeline runs was expressed. For things like image meta-data, this will happen naturally so long as two runs operate on a common input image sequence. However, different pipeline parameters can easily result in varying numbers of difference sources (and therefore objects). Generating the same ids for the same objects/difference sources across runs isn't feasible without significantly complicating the association pipeline. Instead, this type of comparison should be done by performing spatial matches across run databases after runs have finished.

Reducing pipeline database size requirements

With the exception of Object, all known DC3 pipeline tables start off empty and are progressively filled (inserted into) as a run progresses. They are also (with the possible exception of DIASource) relatively small, so optimizations are targeted exclusively at Object.

The first idea is to strip all unused columns from the Object schema - for DC3, this is likely to be a significant fraction. A consequence of this is that association pipeline performance numbers become less meaningful, though implementing an option to store either fat or skinny rows wouldn't be much extra work. Also, subsequent data challenges are likely to shrink the savings this approach affords us to zero.

The next idea is to have nightly processing operate on a shared, read-only copy of Object (containing *prototypical* objects) plus a run specific delta. Non-prototypical objects consist of new and updated objects. There appear to be two basic implementation choices: separating the prototypical and non-prototypical objects into distinct tables or using a single shared Object table augmented with run identifiers.

In either case, minor modifications to the association pipeline storage layer are required - a single SQL script must be changed.

KTL - Is it really necessary to ever query the merged prototypical and delta Object tables? For comparison purposes and for evaluating the output of DC3, isn't just looking at the delta table good enough? Only if we are going to feed the prototype plus delta back into the processing somehow would we really need to query both.

Single Object table

This option extends the Object table with an indexed *runId* column. New objects are tagged with the appropriate run id and prototypical objects have a run id of NULL.

An update of a prototypical object is converted to an insert with updated values (but the same object id). Updates of non-prototypical objects are handled with the usual SQL `UPDATE` statements. Converting these to inserts is possible and might at some point be desirable (it allows examining object attribute evolution over the course of a run), but would require explicit row versioning and additional disk space.

Deletes of prototypical objects are converted to inserts of new objects with a *deleted* flag set to true. Deletes of non-prototypical objects are converted to updates (that set the *deleted* flag to true). Note however that deletes aren't required for DC3.

Many databases (e.g. Postgres) convert SQL `UPDATE`s to "mark row deleted, insert new row" internally. If each object is updated many times, this will obviate the space advantages of the proposed delta scheme unless the pipeline framework is careful to reclaim space occupied by obsoleted rows (e.g. with Postgres, one should `VACUUM` the Object table after run completion).

Implementation:

Lets assume we are executing run 'run01', and that the shared object table is dc3.Object. A private run database still exists, but instead of containing an Object table, it contains the following view:

```
CREATE VIEW Object AS
    SELECT a.* FROM dc3.Object AS a LEFT OUTER JOIN
                    dc3.Object AS b ON a.objectId = b.objectId
    WHERE a.runId IS NULL AND b.runId = 'run01'
       AND b.objectId IS NULL AND (a.runId IS NULL OR a.runId = 'run01');
```

If deletes must be supported, the `WHERE` clause should become

```
   WHERE b.objectId IS NULL AND (a.runId IS NULL OR a.runId = 'run01') AND a.deleted = 0;
```

(assuming that `deleted != 0` indicates a row has been deleted).

Pros:

- Comparing runs may be easier if more than 2 runs are involved in the comparison.

Cons:

- All developer data lumped into one big table - when not using the provided view, it is easier to mistakenly select the wrong data. Also, a screw up or bug can easily affect everyones data.
- Deleting data for a run involves a `DELETE` on a potentially large table. These deletes may not actually reclaim space until `VACUUM` (or its analogue) is performed. This is likely to be much more expensive than a simple `DROP TABLE`.
- Queries on the shared Object table must sift through everyones data.
- Implementation of Views in MySQL is pretty bad, see dbViewsInMySQL for more details.

Note that ?MySQL 5.1+ list partitioning can be used to remove the performance downsides of this approach: simply give each *runId* its own partition. Run initialization would then consist of `ALTER TABLE ... ADD PARTITION` and cleanup would simply be an `ALTER TABLE ... DROP PARTITION` (see the ?partition management documentation for details). This approach would limit the number of simultaneously available runs to 1024, as it is the max number of partitions that MySQL can support. If partitioning is utilized to optimize query performance on the shared Object table, then the maximum number of simultaneously available runs may actually be significantly smaller. Note that Postgres ?has partitioning functionality which can be used to do the same thing.

Per-run Object delta table

With this option, prototypical objects are stored in their own table (without any schema changes). Non-prototypical objects get a different table, further augmented with a *deleted* flag column if deletes must be supported.

All inserts go into the delta table and updates/deletes are converted to inserts and updates much as before. Again, the delta table might need to be `VACUUM`ed after run completion for maximum space savings.

Implementation:

Lets assume the shared object table is dc3.Object. The private run database contains an ObjectDelta table and the following tables/views:

```
-- This pastes together the shared Object table with the per-run delta
CREATE TABLE AllObjects (
    ...,
) ENGINE=MERGE UNION(dc3.Object, ObjectDelta);

-- Note, this view is equivalent to the above, but MySQL processes UNION
-- views by materializing them to temp tables. In fact, even running EXPLAIN
-- on a very simple SELECT over this VIEW simply hangs, with iostat reporting
-- intense but sporadic write activity (and no read activity).
CREATE VIEW AllObjectsView (
    ...,
    runId
) AS SELECT *,0 FROM dc3.Object
  UNION
  SELECT *,1 FROM ObjectDelta;

-- This view causes rows in the delta table to "override" those also present in the read-only table
CREATE VIEW Object AS
    SELECT a.* FROM AllObjects AS a LEFT OUTER JOIN
                    AllObjects AS b ON a.objectId = b.objectId AND b.runId > a.runId
    WHERE b.objectId IS NULL;

-- An alternative suggested by K-T is to use a dependent subquery
CREATE VIEW Object AS
    SELECT * FROM AllObjects AS a
    WHERE runId = (SELECT MAX(runId) from AllObjects WHERE objectId = a.objectId);
```

Pros:

- Cleaning up a run is simple and fast: all that's required is `DROP TABLE`
- Access control can be setup such that developers are unable to write to the shared Object table
- Queries need never touch data from unrelated runs.

- The read-only dc3.Object table can be compressed with ?myisampack

Cons:

- Comparing more than 2 runs at once may be harder than with the previous approach.
- If we use views, implementation of Views in MySQL is pretty bad, see dbViewsInMySQL for more details.

Merge-table based approach with no views

Let's look closer at the merge table approach that does not rely on any views.

First note, that the merge table should not have a primary key, even though the underlying tables do. If it does, our scheme of comparing runs to detect updates won't work (mysql will return first row found and stop searching for more). I noticed doing the "obvious" thing confuses mysql as can be seen below:

```
create table x (id int primary key);
  create table y (id int primary key);
  create table m (id int) ENGINE=MERGE UNION(x,y);

  select * from m;
  ERROR 1168 (HY000): Unable to open underlying table which is
  differently defined or of non-MyISAM type or doesn't exist
```

The way to avoid it is:

```
create table x (id int primary key);
  create table y (id int primary key);
  create table m (id int primary key) ENGINE=MERGE UNION(x,y);
  alter table m drop primary key;
```

Here is an example how one would query such merge table:

```
-- create schema for the ObjectPrototypical (read only)
create table oProt (
  id int primary key,
  n int not null,
  deleted bool default 0,
  runId int
);

-- create schema for ObjectDelta table for run 1
create table oDelta1 like oProt;

-- create merge table
create table o1 (
  id int primary key,
  n int not null,
  deleted bool default 0,
  runId int
) ENGINE=MERGE UNION(oProt, oDelta1);
alter table o1 drop primary key;

-- insert some dummy data to prototypical object table
insert into oProt(id, n, runId) values (1,1,0), (2,9,0), (3,3,0), (7,34,0);
```

```
-- insert some dummy data to run1 specific table
insert into oDelta1 values
 (5,5,0,1),  -- new entry
 (2,2,0,1),  -- overwriting existing entry
 (7,0,1,1);  -- deleting one entry

select * from o1;

+----+----+---------+-------+
| id | n  | deleted | runId |
+----+----+---------+-------+
|  1 |  1 |       0 |     0 |
|  2 |  9 |       0 |     0 |
|  3 |  3 |       0 |     0 |
|  7 | 34 |       0 |     0 |
|  5 |  5 |       0 |     1 |
|  2 |  2 |       0 |     1 |
|  7 |  0 |       1 |     1 |
+----+----+---------+-------+

-- and finally, a more useful query one would run to select objects from run1
-- it takes care of updates and deletes
select * from o1 as a
where runId = (select max(runId) from o1 where id=a.id) and deleted = 0;

+----+---+---------+-------+
| id | n | deleted | runId |
+----+---+---------+-------+
|  1 | 1 |       0 |     0 |
|  3 | 3 |       0 |     0 |
|  5 | 5 |       0 |     1 |
|  2 | 2 |       0 |     1 |
+----+---+---------+-------+
```

Pros:

- views are avoided

Cons:

- introduces self join of large table
- query syntax not completely straightforward

Approach without views and without merge table

To avoid the extra self join and views all together, we can try the following:

```
-- this is the read only prototypical table
create table o (
  id int primary key,
  n int not null,
  s char default NULL -- 'd': deleted, 'u': updated, 'n': new
);

-- this is the per run delta table
create table o1 like o;
```

```
insert into o(id, n) values (1,1), (2,9), (3,3), (7, 34);

insert into o1 values (5,5, 'n'), (2, 2, 'u'), (7, 0, 'd');


select * from o;

+----+----+------+
| id | n  | s    |
+----+----+------+
|  1 |  1 | NULL |
|  2 |  9 | NULL |
|  3 |  3 | NULL |
|  7 | 34 | NULL |
+----+----+------+

select * from o1;

+----+---+------+
| id | n | s    |
+----+---+------+
|  5 | 5 | n    |
|  2 | 2 | u    |
|  7 | 0 | d    |
+----+---+------+


And the real query that select objects from run 1:

select o.* from o  left join o1 USING(id) where o1.s is null
  union
select * from o1 where s <> 'd';

+----+---+------+
| id | n | s    |
+----+---+------+
|  1 | 1 | NULL |
|  3 | 3 | NULL |
|  5 | 5 | n    |
|  2 | 2 | u    |
+----+---+------+
```

Pros:

- self-join of the big table avoided, we do a join with delta table which is much smaller

Cons:

- query syntax is still not completely straightforward

Thought: perhaps we could automatically rewrite a query, for example user types a query:

```
SELECT Object.<someColumns>, Source.<someSolumns>
FROM Object
JOIN Source USING (objectId)
WHERE Source.x = <x> and Object.y = <y>
GROUP BY <whatever>
```

and we automatically translate it into:

```
SELECT Object.<someColumns>, Source.<someSolumns>
FROM Object
JOIN Source USING (objectId)
LEFT JOIN Object_<runNumber> USING (objectId)
WHERE Source.x = <x> and Object.y = <y>
  AND Object_<runNumber>.s IS NULL
  UNION
SELECT Object_<runNumber>.<someColumns>, Source.<someSolumns>
FROM Object_<runNumber>
JOIN Source USING (objectId)
WHERE s <> 'd'
GROUP BY <whatever>
```

More Aggressive Run Database Cleanup

During DC2, per-run databases were deleted using a "delete a run when its creator remembers to do so" policy. Considering that the debugging process often resulted in tens of databases for failed runs, each containing a complete copy of the input Object catalog, this seems like a good (and easy) target for improvement.

I propose we keep a database table which contains the following fields for each run:

- run database name
- the time at which the run was started
- transient flag

Run initialization would be modified to include the following steps

- perform cleanup actions for every run in the global run table that is marked transient and is older than X days.
- insert run information into the global run table, setting the transient flag to true.

Developers would also be provided with a trivial convenience script to "bless" a run - this script would set the transient flag for the run to false, thereby preventing its data from being automatically dropped.

Note that performing cleanup immediately prior to launching a run ensures that the run has access to as much free space as possible, and avoids run/cleanup database contention that could occur when cleanup is scheduled to occur periodically. On the other hand, run initialization time could be adversely affected.

TODO

- ~~Try to figure out a better way than `UNION` views to implement the delta table approach.~~ (Use a `MERGE` table, thanks K-T!)
- Investigate the impact of the proposed views on queries that can be posed.
- Investigate query performance on the proposed views (in particular, it would be good to have reasonably fast spatial matching capabilities).

Conclusion

If partitioning functionality is unavailable, storing run specific data in a separate table seems like the better solution. Otherwise, the approaches appear relatively similar from a performance point of view.