PostgreSQL Perfomance

Tables partitioning vs. Aggregated data tables

Introduction

Here’s a classic scenario. You work on a project that stores data in a relational database. The application gets deployed to production and early on the performance is great, selecting data from the database is snappy and insert latency goes unnoticed. Over a time period of days/weeks/months the database starts to get bigger and queries slow down.

Potential solutions

A Database Administrator (DBA) will take a look and see that the database is tuned. They offer suggestions to add certain indexes, move logging to separate disk partitions, adjust database engine parameters and verify that the database is healthy. This will buy you more time and may resolve this issues to a degree.

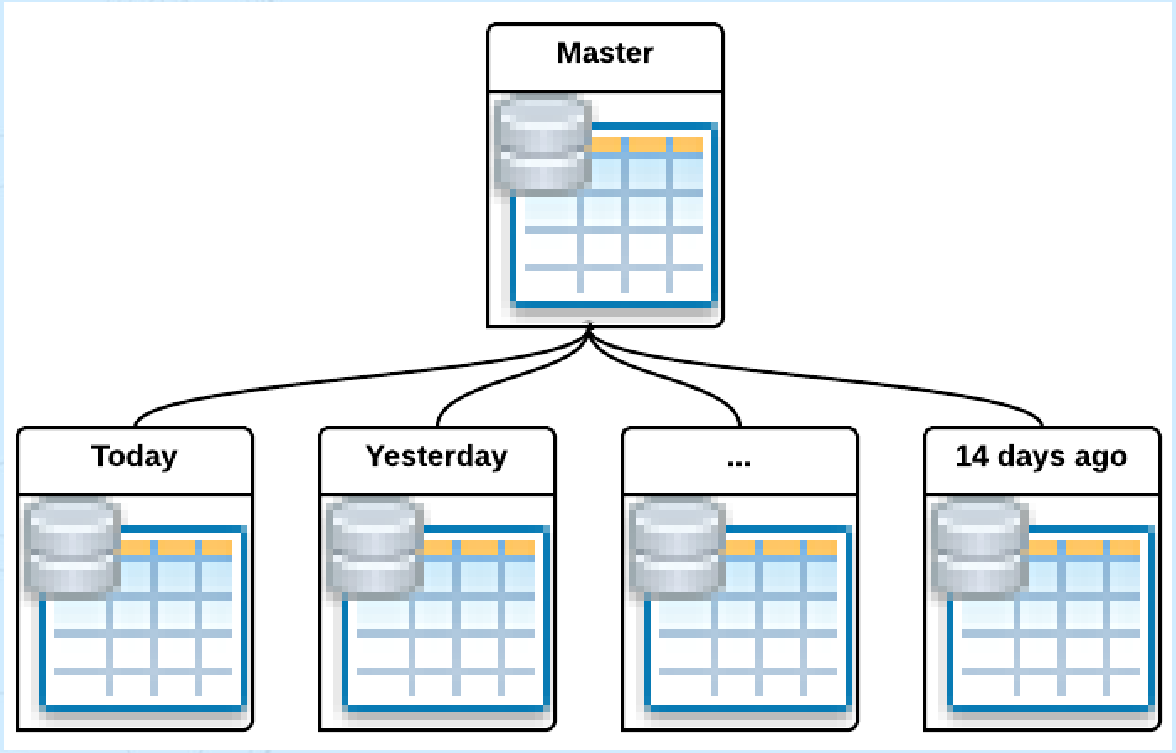
At a certain point you realize the data in the database is the bottleneck.

There are various approaches that can help you make your application and database run faster. Let’s take a look at two of them:

- Table partitioning
- Aggregated data tables

Table Partitioning

Main idea: you take one massive table (**master table**) and split it into many smaller tables – these smaller tables are called **partitions** or **child tables**.



Terminology

Master Table
Also referred to as a **Master Partition Table**, this table is the template child tables are created from. This is a normal table, but it doesn't contain any data and requires a trigger.

Child Table
These tables inherit their structure (in other words, their **Data Definition Language** or **DDL** for short) from the master table and belong to a single master table. The child tables contain all of the data. These tables are also referred to as **Table Partitions**.

Partition Function
A partition function is a Stored Procedure that determines which child table should accept a new record. The master table has a trigger which calls a partition function.

How it can be implemented?

Here’s a summary of what should be done:

1. Create a master table
2. Create a partition function
3. Create a table trigger

Let’s assume that we have a rather large table (~ 2 500k rows) containing reports for different dates.

Partition Function

There are two typical methodologies for routing records to child tables:

- By Date Values
- By Fixed Values

The trigger function does the following:
Creates child table by dynamically generated “CREATE TABLE” statement if the child table does not exist.

Partitions (child tables) are determined by the values in the “date” column. One partition per calendar month is created.

The name of each child table will be in the format of “master_table_name_yyyy-mm”

Partition Function

```
CREATE OR REPLACE FUNCTION partition_function() RETURNS trigger AS
$BODY$
DECLARE
                table_master    varchar(255)  := ‘SOME_LARGE_TABLE';
                table_part varchar(255) := ‘';
…
BEGIN
-----------------------------------------generate partition name-----------------------------------------------
…
table_part := table_master|| '_y' || DATE_PART( 'year', rec_date )::TEXT
                                             || '_m' || DATE_PART( 'month', rec_date )::TEXT;
----------------------------------------check if partition already exists---------------------------------------------
…
----------------------------------------if not yet then create new-----------------------------------------------------
EXECUTE 'CREATE TABLE public.' || quote_ident(table_part) || ' (
CHECK( “RECORD_DATE" >= DATE ' || quote_literal(start_date) || ' AND “RECORD_DATE" <
DATE ' || quote_literal(end_date) || ')) INHERITS ( public.'  || quote_ident(table_master) ||  ')
----------------------------------------create indexes for current partition-----------------------------------------
EXECUTE 'CREATE INDEX …
END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
```

Table Trigger

Now that the Partition Function has been created an Insert Trigger needs to be added to the Master Table which will call the partition function when new records are inserted.

```
CREATE TRIGGER insert_trigger
BEFORE INSERT
ON “SOME_LARGE_TABLE"
FOR EACH ROW
EXECUTE PROCEDURE partition_function();
```

At this point you can start inserting rows against the Master Table and see the rows being inserted into the correct child table.

Constraint Exclusion

Constraint exclusion is a query optimization technique that improves performance for partitioned tables

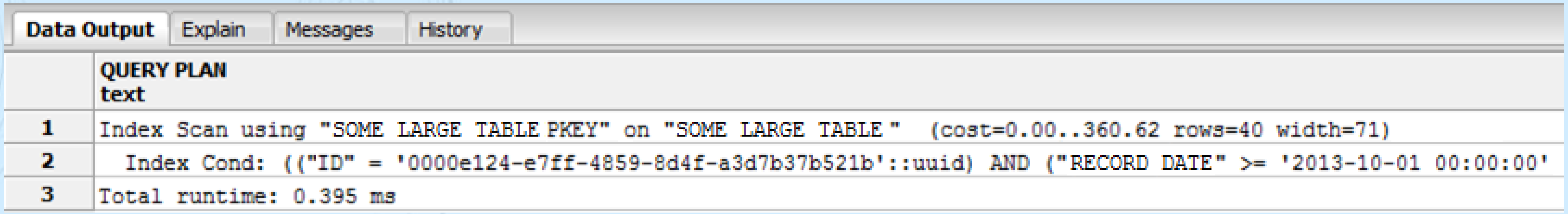
```
SET constraint_exclusion = on;
```

The default (and recommended) setting of constraint_exclusion is actually neither on nor off, but an intermediate setting called *partition*, which causes the technique to be applied only to queries that are likely to be working on partitioned tables. The on setting causes the planner to examine CHECK constraints in all queries, even simple ones that are unlikely to benefit.

Constraint Exclusion

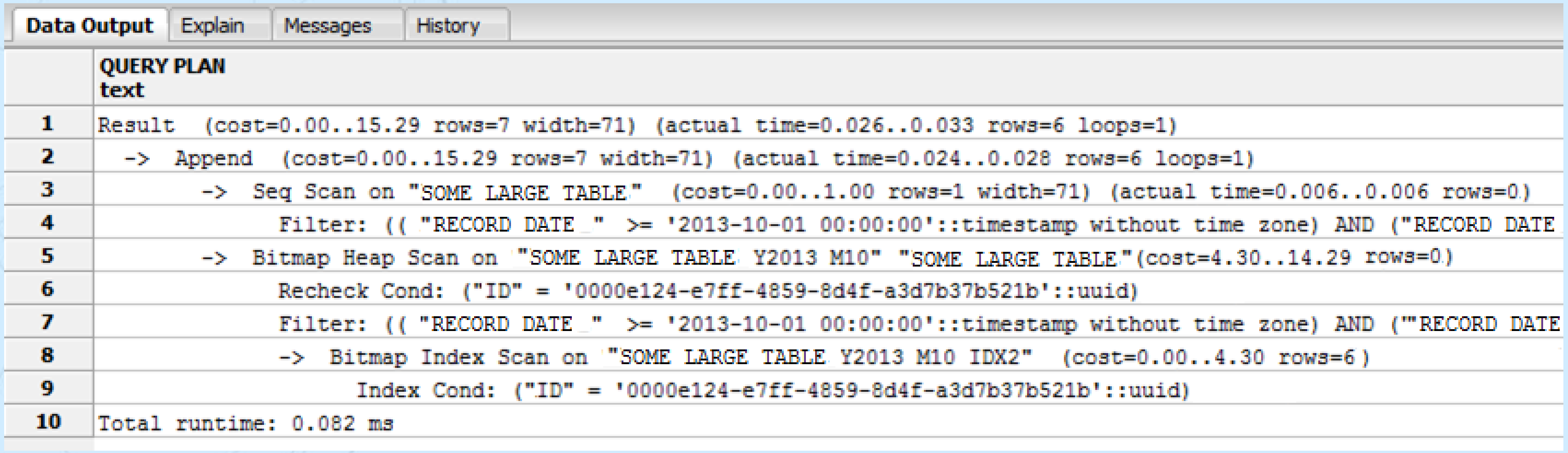
SELECT * FROM “SOME_LARGE_TABLE" WHERE “ID" = '0000e124-e7ff-4859-8d4f-a3d7b37b521b' AND “RECORD_DATE" BETWEEN '2013-10-01' AND '2013-10-30';

Without partitioning:



	QUERY PLAN text
1	Index Scan using "SOME LARGE TABLE PKEY" on "SOME LARGE TABLE " (cost=0.00..360.62 rows=40 width=71)
2	Index Cond: (("ID" = '0000e124-e7ff-4859-8d4f-a3d7b37b521b'::uuid) AND ("RECORD DATE" >= '2013-10-01 00:00:00'
3	Total runtime: 0.395 ms

With partitioning:



```
Result  (cost=0.00..15.29 rows=7 width=71) (actual time=0.026..0.033 rows=6 loops=1)
  ->  Append  (cost=0.00..15.29 rows=7 width=71) (actual time=0.024..0.028 rows=6 loops=1)
        ->  Seq Scan on "SOME LARGE TABLE"  (cost=0.00..1.00 rows=1 width=71) (actual time=0.006..0.006 rows=0)
              Filter: (( "RECORD DATE _"  >= '2013-10-01 00:00:00'::timestamp without time zone) AND ("RECORD DATE
        ->  Bitmap Heap Scan on  "SOME LARGE TABLE  Y2013 M10"  "SOME LARGE TABLE"(cost=4.30..14.29 rows=0)
              Recheck Cond: ("ID" = '0000e124-e7ff-4859-8d4f-a3d7b37b521b'::uuid)
              Filter: (( "RECORD DATE _"  >= '2013-10-01 00:00:00'::timestamp without time zone) AND (""RECORD DATE
              ->  Bitmap Index Scan on  "SOME LARGE TABLE  Y2013 M10 IDX2"  (cost=0.00..4.30 rows=6 )
                    Index Cond: ("ID" = '0000e124-e7ff-4859-8d4f-a3d7b37b521b'::uuid)
Total runtime: 0.082 ms
```

Benefits:

- Query performance can be improved dramatically in certain situations;
- Bulk loads and deletes can be accomplished by adding or removing partitions;
- Seldom-used data can be migrated to cheaper and slower storage media.

Caveats:

- Partitioning should be organized so that queries reference as few tables as possible.
- The partition key column(s) of a row should never change, or at least do not change enough to require it to move to another partition.
- Constraint exclusion only works when the query's WHERE clause contains constants.
- All constraints on all partitions of the master table are examined during constraint exclusion, so large numbers of partitions are likely to increase query planning time considerably.

Aggregated Data Tables

Another approach to boost performance is using pre-aggregated data.
One real feature of relational databases is that complex objects are built from their atomic components at runtime, but this can cause excessive stress if the same things are being done, over and over.

Without using pre-aggregated data you may see unnecessary repeating large-table full-table scans, as summaries are computed, over and over.

Data aggregation can be used to pre-join tables, presort solution sets, and pre-summarize complex data information. Because this work is completed in advance, it gives end users the illusion of instantaneous response time.

How it can be implemented?

You can use a set of ordinary tables with triggers and stored procedures for these purpose but there is another solution available out of the box – materialized views (PostgreSQL v. 9.3 natively supports materialized views)

A *materialized view* is a database object that contains the results of a query Materialized views in PostgreSQL use the rule system like views do, but persist the results in a table-like form.

Let’s assume that we have a two tables: ‘machines’ (2 abstract machines) and ‘reports’ containing reports for each machine (~100k rows).

Create Materialized View

Let’s create materialized view:

```
CREATE MATERIALIZED VIEW mvw_reports AS
SELECT reports.id, machines.name || ' ' || machines.location AS
machine_name, reports.reports_qty
FROM reports
INNER JOIN machines ON machines.id = reports.machine_id;
```

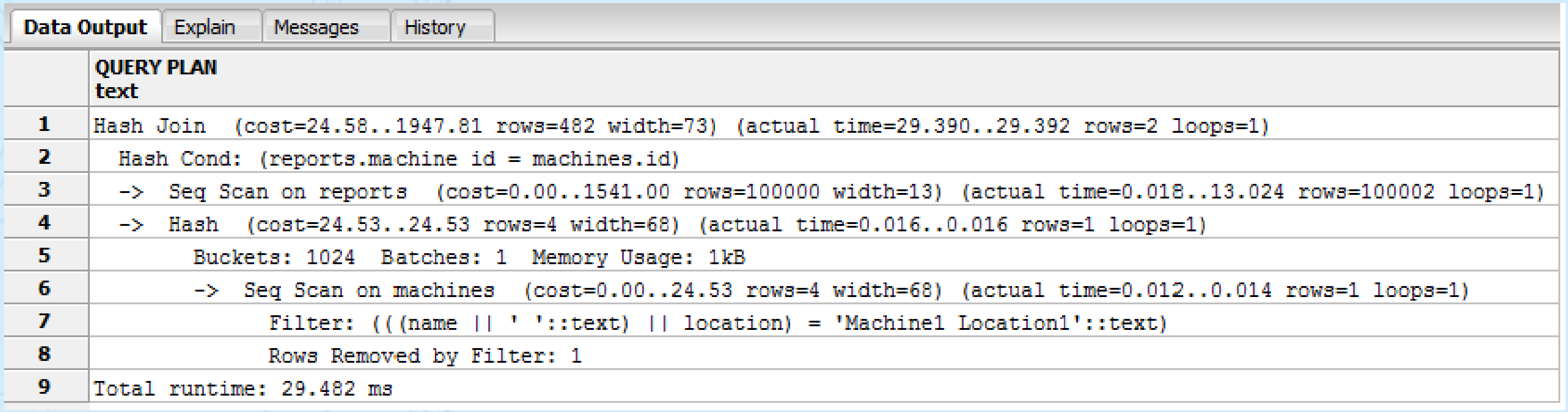
And a simple view for comparison:

```
CREATE  VIEW vw_reports AS
SELECT reports.id, machines.name || ' ' || machines.location AS
machine_name, reports.reports_qty
FROM reports
INNER JOIN machines ON machines.id = reports.machine_id;
```

Create Materialized View

Executing the same query to simple view:

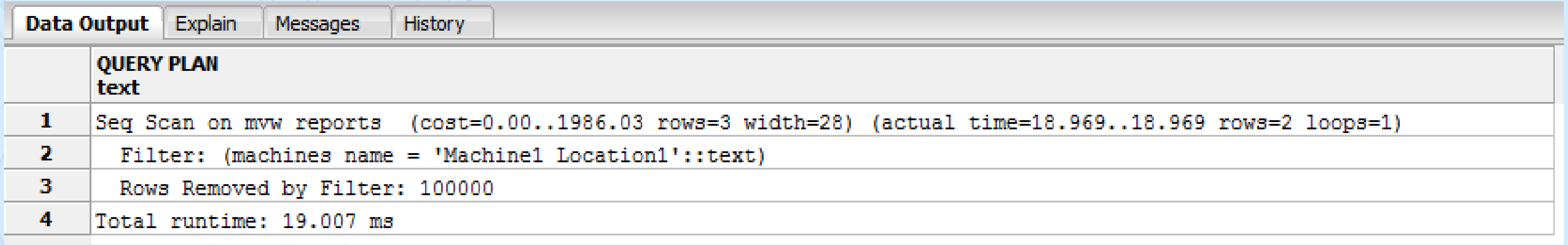
EXPLAIN ANALYZE SELECT * FROM vw_reports WHERE machines_name = ‘Machine1 Location1';



```
QUERY PLAN
text
Hash Join  (cost=24.58..1947.81 rows=482 width=73) (actual time=29.390..29.392 rows=2 loops=1)
  Hash Cond: (reports.machine id = machines.id)
  ->  Seq Scan on reports  (cost=0.00..1541.00 rows=100000 width=13) (actual time=0.018..13.024 rows=100002 loops=1)
  ->  Hash  (cost=24.53..24.53 rows=4 width=68) (actual time=0.016..0.016 rows=1 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 1kB
        ->  Seq Scan on machines  (cost=0.00..24.53 rows=4 width=68) (actual time=0.012..0.014 rows=1 loops=1)
              Filter: (((name || ' '::text) || location) = 'Machine1 Location1'::text)
              Rows Removed by Filter: 1
Total runtime: 29.482 ms
```

And for materialized view:

EXPLAIN ANALYZE SELECT * FROM mvw_reports WHERE machines_name = ‘Machine1 Location1';



```
QUERY PLAN
text
Seq Scan on mvw reports  (cost=0.00..1986.03 rows=3 width=28) (actual time=18.969..18.969 rows=2 loops=1)
  Filter: (machines name = 'Machine1 Location1'::text)
  Rows Removed by Filter: 100000
Total runtime: 19.007 ms
```

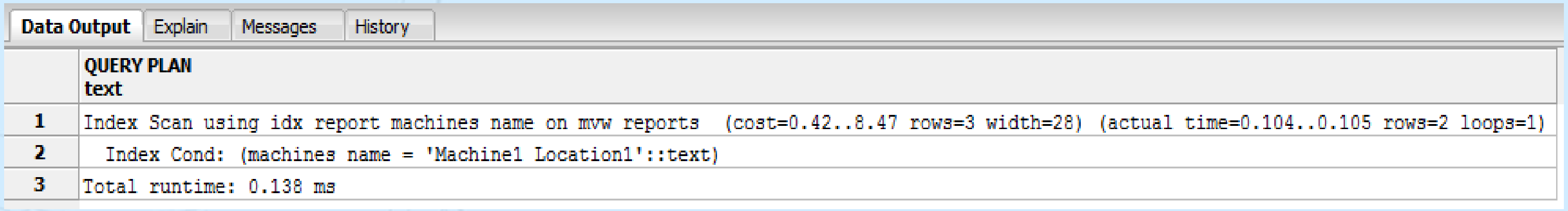
Adding Indexes

Another advantage compared with simple views is that we can add indexes to materialized views like for ordinary tables.

CREATE INDEX idx_report_machines_name ON mvw_reports (machines_name);

Executing the query once more:

EXPLAIN ANALYZE SELECT * FROM mvw_reports WHERE machines_name = ‘Machine1 Location1';



	QUERY PLAN text
1	Index Scan using idx report machines name on mvw reports (cost=0.42..8.47 rows=3 width=28) (actual time=0.104..0.105 rows=2 loops=1)
2	Index Cond: (machines name = 'Machine1 Location1'::text)
3	Total runtime: 0.138 ms

Refreshing materialized view

In order to have actual data in materialized view it should be refreshed after each DML operation (INSERT, UPDATE, DELETE) on the target tables.

```
REFRESH MATERIALIZED VIEW mvw_reports;
```

This can be done using triggers:

```
CREATE TRIGGER machines_refresh AFTER INSERT OR UPDATE OR DELETE ON
machines FOR EACH STATEMENT EXECUTE PROCEDURE mvw_reports_refresh( );

CREATE TRIGGER reports_refresh AFTER INSERT OR UPDATE OR DELETE ON
reports FOR EACH STATEMENT EXECUTE PROCEDURE mvw_reports_refresh ( );
```

Refreshing materialized view

Benefits:

Query performance can be improved dramatically in situations when there are relatively few data modifications compared to the queries being performed, and the queries are very complicated and heavy-weight.

Caveats:

- Materialized views contain a duplicate of data from base tables;
- Depending on the complexity of the underlying query for each MV, and the amount of data involved, the computation required for refreshing may be very expensive, and frequent refreshing of MVs may impose an unacceptable workload on the database server.

Conclusions

Table partitioning and aggregated data tables can help a lot. But there is no ideal solution that always works. Both approaches have their own pluses and minuses. It all depends on certain situation and circumstances. Hopefully presented overview gave few tips on when each technique can be useful.

Any questions?