Scalability Engineering: Autopartitioning

D3.5 v1.0

WP3 – Scalability Engineering: D3.5 Autopartitioning

Dissemination Level: Confidential

Lead Editor: Alvaro Herrera

Date: 31/10/2014

Status: Final

Description of Work:

T3.3 Implement auto partitioning as a standalone feature. Autopartitioning would allow the database to automatically collect statistics that it can use to allow queries to avoid accessing certain parts of a table. Partition stats would then be usable automatically for access by any column, not just pre-defined partition key columns. The technique will react intelligently as data values change since the statistics adjust to the location of the data, rather than have the data placement be driven by the requirements of a pre-defined layout.

Contributors:

Alvaro Herrera

Internal Reviewer(s):

UL: Janez Demsar
UNIMAN: Javier Navaridas

Version History

Version	Date	Authors	Sections Affected
0.1	25/10/2014	Alvaro Herrera	Initial version
	29/10/2014	Alvaro Herrera	Revisions following internal review
1.0	31/10/2014	Simon Riggs	Revisions following review

Table of Contents

1. Summary	3
2. Analysis/ Description of work done	3
3. Evaluation	6
4. Conclusion/ Future	7
5. References	8

List of Figures

Figure 1 : Data Loading, Performance and Scalability	7
Figure 2 : Comparison of Elapsed Times for Index Creation	7

Abbreviations

BRIN	Block Range Indexes

1. Summary

Block Range Indexes, or BRIN for short, provide a low-maintenance-cost mechanism to speed up queries that scan extremely large tables. This can be seen as similar to partitioning the table in the way that the correlation of data within the table is used to avoid scanning sections of the table. This feature delivers large speedups for many queries without incurring the large maintenance costs of traditional indexing approaches, and without imposing the complexity of traditional partitioning schemes that require data model changes. In short, this novel feature makes it possible to use PostgreSQL in a different way than was previously possible. We present some initial evaluations of our results.

BRIN is in some ways similar to work done in other databases. AXLE has extended this with an innovative new approach; we have created a much more generic framework that will allow this work to apply to differing datatypes and a variety of use cases.

BRIN code is now published to the PostgreSQL project. The current version, v20 has received final review comments and is expected to be committed in November 2014.

http://www.postgresql.org/message-id/20141006223359.GL7043@eldon.alvh.no-ip.org

The BRIN feature will be available in the production release of PostgreSQL 9.5, due for release in Sept 2015.

2. Analysis/ Description of work done

The project name has changed a number of times during development, as our understanding of the requirements and implementation has grown.

This project started with an interest in automatically separating data into partitions as it came into a table, so that the most common queries would be sped up by allowing them to skip reading partitions according to query conditions; hence the original name autopartitioning.

Once we got to the detailed design phase, we decided that a new index type was the most appropriate way of implementing this; at the time we were only considering the minimum and maximum values for one-dimensional data types with natural sort orders, so we renamed the project “Minmax indexes”.

Eventually, in the course of community review discussions it became apparent that the design could be generalized to support arbitrary summary information, enabling completely different datatypes to be indexed as well, at which point we again renamed the feature to Block Range Indexes, or BRIN for short, which is what it is today.

We settled on a design that requires that the user declares exactly which columns to track summary values for, by way of a CREATE INDEX command. An example of the usage against the TPC-H database would be

CREATE INDEX ON lineitem *USING BRIN* (l_shipdate);

No other user requests need by made - index usage is completely automatic, given selection by the PostgreSQL cost-based optimizer for specific queries.

The USING clause is a standard PostgreSQL construct allowing the user to specify one of the many index types available, such as BTREE, HASH, GIST or GIN, now also BRIN.

BRIN works at index creation time by scanning a table in portions at a time, each of which we call a “block range”. For each block range the index stores some summary information. As the table is updated by insertion of new records, the summary data is kept up to date automatically, just like a regular index.

When we run queries using the index the summary information is compared to the WHERE conditions in the query. If the conditions are implied by the summary information then we need to examine all the pages within that range, and for each page return the rows that match the query conditions. Conversely, if the summary information does not match the conditions in the WHERE, we know it is not possible for there to be any rows of interest and thus we can skip reading the pages in that range altogether.

BRIN uses the standard facilities for BitmapIndexScan. Queries return a bitmap, which can then be ANDed or ORd alongside other indexes to refine a set of blocks to scan. Rows are then accessed using a BitmapHeapScan. An example of usage can be seen with this extract of an EXPLAIN command of Q4 from the TPC-H

```
Bitmap Heap Scan on orders
  (cost=6127.12..268564.24 rows=577475 width=20)
  Recheck Cond:((o_orderdate >= '1995-05-01'::date) AND
                (o_orderdate < '1995-08-01'::date))
  Rows Removed by Index Recheck: 14426334
  Heap Blocks: lossy=253567
  Buffers: shared hit=253640
          ->  Bitmap Index Scan on orders_o_orderdate_idx
                (cost=0.00..5982.75 rows=577475 width=0)
                Index Cond: ((o_orderdate >= '1995-05-01'::date) AND
                             (o_orderdate < '1995-08-01'::date))
                Buffers: shared hit=73
```

The bitmap is a “lossy” bitmap, meaning we return only the *blocks* in the heap that need to be scanned. So we must recheck the *rows* using the original index condition.

The full implementation ensures that BRIN indexes are both crash-safe and concurrent, so after testing we are confident that BRIN is completely ready for wide dissemination.

Competing database systems (such as Oracle Exadata in its “storage indexes”, Netezza in its “zone maps”, Infobright in its “data packs” and MonetDB) have a very similar feature, but they restrict it to minimum and maximum values only.

BRIN does not restrict the summary information to just minimum and maximum values, but instead provide a programmable mechanism that allows different storage depending on the data type as well as the user's knowledge of data distribution. The set of operations necessary to implement summarization of any individual data type is called an “operator class”. Storing minimum and maximum values is one of the possible operator class designs, and it enables the feature to be used for data types that have a natural sort order, such as integer numbers, real numbers and dates, among others. The code contribution to PostgreSQL contains operator classes for all such data types that are included in core PostgreSQL covering the SQL Standard data types.

Additionally, it is possible to construct operator classes for data types as varied as geometric and geographic data types (by storing the bounding box common to all elements in a page range). We have assisted and advised other PostgreSQL community members to implement BRIN indexes for other data types, such as inet and range datatypes. Development for new datatypes is not trivial, but this shows it is both desirable and possible.

The generality of the design, as far as we know, is an innovation unique to PostgreSQL and a clear advance beyond the state of the art as a result of the AXLE project.

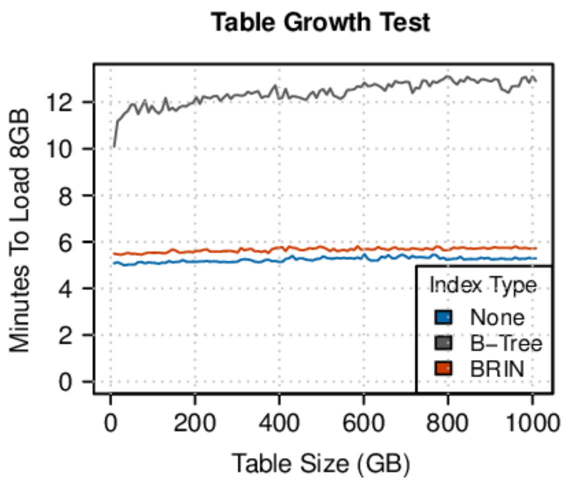
We have performed initial analysis on the current BRIN implementation, which is covered in the next section. Wider evaluation will be published as part of the holistic benchmark evaluations.

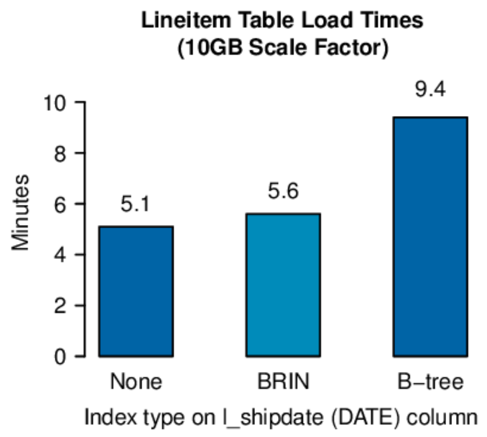
3. Evaluation

One advantage of this new type of index is the low maintenance cost. In PostgreSQL, there are very fast techniques for scanning B-Tree indexes, even when the returned row set is very large; so arguably it would work to use B-Tree indexes for all columns in a table that are used in the WHERE clause of a query. However, the cost of maintaining a B-Tree index as the table is inserted into or modified is very high and most importantly the cost grows as the table gets larger.

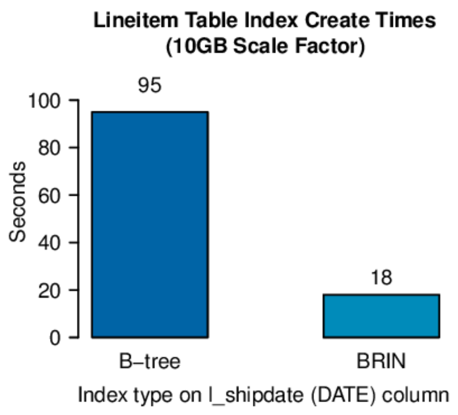
BRIN indexes imposes a much lower cost on new insertions, plus we have validated (Figure 1) that the cost is constant no matter the size of the table. BRIN allows many more columns than would normally be considered and/or on very large tables.

Figure 1: Data Loading, Performance and Scalability on Large Databases





The main use case for AXLE is the case of continuously updated databases being accessed for analytical purposes. For this use case, it is just not possible to create all the B-Tree indexes that one would otherwise; in addition B-Tree index creation can reduce the resource available for analysis for long periods. BRIN index create times are very fast.



One of the main reasons for the much faster build times are that the BRIN index relies upon the natural sorted order of the data. BRIN is only effective when columns with a natural correlation between their data values and the physical table ordering. Examples of where this could help would be both OrderId and OrderDate.

The second aspect working to reduce index build time is simply size - the BRIN index is orders of magnitude smaller. Our tests show that on a 10GB table, a B-Tree index would be around 1GB, whereas the BRIN index is just 1.7MB, some 612 times smaller.

Index performance has not gained significantly by using BRIN, though neither has it been measured to decrease significantly against B-Trees that access many rows, as would be required in analytical queries.

We hope for significant performance gains in very large databases due to the reduced I/O associated with reading indexes, using the same “I/O avoidance” theme AXLE is also pursuing with column storage techniques. BRIN will work with column stores, given the design that is now emerging. We also expect residual benefit from better use of available RAM as a result of the reduced size of the BRIN index.

With BRIN, a single index can cover many columns of a table with essentially the same very low insertion cost. It becomes feasible to have a BRIN index that covers enough columns necessary to satisfy many more queries, and yet not have it become an excessive burden performance-wise.

3. Conclusion/ Future

We have succeeded at providing a low-cost and extensible mechanism to optimize scans of very large tables, something that traditional indexes cannot do because of the high maintenance cost that comes with them. Applications in which PostgreSQL could previously not be a viable competitor due to performance limitations will now be able to use BRIN, and some existing users can expect performance improvements.

In particular, users that are required to preserve historical data for long periods will be able to keep that data online, without expensive or complicated archive and prune procedures to get rid of old data.

It is expected that novel uses of the generic per-block-range summary information provided by BRIN will be found and we will work with potential users to create novel uses. Additional uses so far suggested have been use for Bloom filter indexes and as a way of indexing large stores of GIS data.

4. References

Other database systems already have similar features. Some examples:

* Oracle Exadata calls this "storage indexes"

http://richardfoote.wordpress.com/category/storage-indexes/

* Netezza has "zone maps"

http://nztips.com/2010/11/netezza-integer-join-keys/

* Infobright has this automatically within their "data packs" according to a

May 3rd, 2009 blog post

http://www.infobright.org/index.php/organizing_data_and_more_about_rough_data_contest/

* MonetDB also uses this technique, according to a published paper

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.108.2662

"Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS"