Collaborative Project

LOD2 – Dynamic Repartitioning

Project Number: 257943 **Start Date of Project:** 01/09/2010 **Duration:** 48 months

Deliverable 2.2

Dynamic Repartitioning

Dissemination Level	Public
Due Date of Deliverable	Month 12, 31/08/2011
Actual Submission Date	31/08/2011
Work Package	WP 2, Storing and Querying Very Large Knowledge bases
Task	T2.2
Type	Software
Approval Status	Approved
Version	1.0
Number of Pages	9
Filename	LOD2_D2.2_Dynamic_Repartitioning

Abstract:

This report gives an overview of the new Virtuoso Elastic Cluster (VEC) enhancements allowing the creation of arbitrarily large databases that can be spread across a variable number of servers for dynamic growth and repartitioning of the cluster.

Project co-funded by the European Commission within the Seventh Framework Programme (2007 – 2013)

History

Version	Date	Reason	Revised by
0.1	29/08/2011	Draft of initial Dynamic repartitioning document	Orri Erling
0.5	30/08/2011	Peer-review comments	Giovanni Tummarello
0.7	30/08/2011	Approval comments	Peter Boncz
1.0	31/08/2010	Final revision	Hugh Williams Orri Erling

Author List

Organisation	Name	Contact Information
OLG	Hugh Williams	hwilliams@openlinksw.com
CWI	Peter Boncz	P.Boncz@cwi.nl
NUIG	Giovanni Tummarello	giovanni.tummarello@deri.org
OLG	Orri Erling	oerling@openlinksw.com

Table of Contents

Elastic Virtuoso Cluster

Introduction

Virtuoso Elastic Cluster (VEC) allows creating arbitrarily large databases that can be spread across a variable number of servers. As the database grows, servers may be added in order to provide enough CPU, memory and storage for meeting demand as it occurs. If there are large fluctuations in database utilization, extra capacity may be provisioned on demand and later freed when no longer needed.

A Virtuoso cluster database consists of multiple logical clusters. Each logical cluster consists of one or more host groups. The data is partitioned among the host groups. A host group consists of a single host managing the partition or may optionally consist of multiple hosts managing identical copies of said partitions. Such a host group is also called a quorum.

Every database contains minimally two logical clusters, called REPPLICATED and ALL. All schema tables are allocated in the REPLICATED logical cluster, which consists of a single partition with a copy on each host. Most application tables are allocated in the ALL logical cluster which partitions the data among the host groups in the cluster. The partitioning criteria are specified key by key.

A VEC enabled (elastic) cluster database makes the ALL logical cluster elastic.

A fixed (non-elastic) logical cluster has one partition per host group (aka quorum).

An elastic logical cluster has a variable number of partitions that can be moved between host groups.

This also means that host groups may be added to the elastic logical cluster and removed from it during the cluster's lifetime.

An elastic cluster may store partitions in multiple copies, that is, each host group in the cluster may consist of several mutually replicating hosts. An object, e.g. table or index is assigned to a logical cluster by the alter index statement or the partition clause in the create index statement. If the logical cluster specified is elastic, the data belonging to the object will be managed as a number of relatively small partitions that can easily migrate between host groups, thus offering convenient repartitioning as data scale changes.

Partitioning Principles

An index may have one or more partitioning columns. Each partitioning column specifies what part of the column's value is used for computing the partitioning hash. This may be a bit mask for an integer-like value or a specification of using n leading characters or all except n trailing characters for a string valued column.

The partitioning hash designates a logical slice into which the index entry falls. There is a preset number of logical slices, e.g. 16K, which also forms the maximum of physical slices. One or more logical slices are mapped into a physical slice. A physical slice is assigned to a host group. Each host in the host group maintains identical copies of the physical slice.

When a physical slice grows past a given size, e.g. 10GB, it can be split into two physical slices, with one half of the logical slices going to one side and the other half to the other.

In this manner, no partition grows past a given size limit, meaning that the partitions remain of a conveniently moveable size. Thus, if 10GB is the maximum conveniently moveable size, e.g. copyable in about 2-5 minutes over 1GbE) and there were 16K logical slices, the database could grow to 160TB before partitions started growing past the 10GB limit, there being no more logical slices available.

Of course, since the logical slice number is just a bit field from a longer hash number, there is no hard limit on the number of logical slices.

A physical slice is a self-contained Virtuoso database file or set of database file stripes. It contains a shard of all the objects created into the logical cluster of which this is a part. This does not contain schema tables since these are always created in the REPLICATED logical cluster.

The logical slice has its own checkpoint state. A checkpoint state is a consistent read-only image of the database. A page that is changed after the checkpoint is mapped to an alternate delta page that will be merged into the checkpoint'ed state on the next log checkpoint.

Logical slices managed by a server process share one transaction log. Upon roll forward, the system calculates for each partitioned object which slice it will hall into.

Since the log and checkpointing are interrelated (a new log is started or the old log is reused after a checkpoint), all slices maintained by a server are checkpointed together, in parallel.

Splitting and Motion of Partitions

Splitting and motion of partitions introduce special restrictions into the cycle of log checkpoints on the concerned server. Generally, each server process in a cluster can checkpoint independently of the others. In practice, synchronizing checkpoints is useful since checkpointing introduces some slowdown due to a peak of IO for writing out dirty buffers and a small atomic window for syncing the persistent state on disk.

A physical slice can split by first making a checkpoint only on the slice in question. This checkpoint does not reuse the log but instead records a level in the current log and prohibits making a global checkpoint during the time interval of the split. Many splits may proceed independently in parallel.

During the split, the checkpointed state of the slice is read and written into two halves. This is a linear time operation since every index in the slice is read and written once, in key order. While this is proceeding, the original slice can continue to be updated. Once the split(s) are complete, the log is replayed on top of the split halves starting at the position recorded when the slice was checkpointed. When the first of the splits is in sync, a cluster-wide atomic section begins, with the following operations:

1. Request atomic, exclusive access to all hoists in the cluster. If all hosts that are supposed to be online do not acknowledge this within a timeout period, the slit aborts and the results are discarded. This can happen for example if a host failure coincides with the completion of the split and is infrequent.

This has the effect of aborting all uncommitted transactions. This eliminates the need to move uncommitted state, locks and states of query executions that may be waiting for the locks between slices, which is overly complex even though theoretically conceivable.

2. With atomic access to the whole cluster, the cluster map (mapping of logical slices to physical slices and mapping of physical slices to host groups) is updated.

3. After all hosts have confirmed applying the update the atomic section finishes and normal operation resumes.

When a physical slice is moved between host groups, a process similar to a slice split takes place. The data is copied in the background, based on a checkpointed state, after which the relevant log records are shipped to the recipient. When the log is almost in sync, an atomic section such as the one described above takes place.

In the event of partitions being maintained in duplicate, splits and motion occur simultaneously on all copies. If a member of the host group (quorum) is not available, splits are not allowed in the interest of avoiding complex special cases.

Since checkpoints are prevented and a rollback of uncommitted transactions is involved, it makes sense to do many splits at one time in order to minimize the interference with online operation.

Interaction of Partitioning and Compression

Compression relies on similar values occurring together. Due to this, the partitioning hash does not typically consider the low 8 or 16 bits of identifier data types (i.e. integer, IRI ID, RDF literal ID).

With RDF data, there is great variability in the distribution of the object values, e.g. certain RDFS classes or other enumeration-like property values are much more common than others.

Thus when partitioning by a hash of the high bits O, the partition sizes may be uneven. This is compensated for by two factors:

1. With many repetitions of the same value, compression becomes more effective, e.g. run length for first key part, densely ascending for 2nd key part etc.

2. IRI ID's and RDF literal ID's are allocated from a range of id's that are chosen based on a hash of the string representation. Thus values consecutively allocated are likely not to share their high bits since the ranges differ just in the high bits. Within the ranges id's are allocated consecutively but all values are first hashed into a large number of possible ranges, so that the likelihood of two common values getting the same range, hence the same slice, is diminished.

Usage Statistics and Partition Distribution

Databases often exhibit temporal locality, data that are inserted together are accessed together and more recently inserted data is more likely to get accessed.

With RDF, these patterns still occur at least in some orderings of the data but the matter is less clear-cut than in most relational situations.

Still, for purposes of getting the most rapid overall response from redistribution of data, it is important to track usage patterns. Besides, moving a slice that is seldom used is unlikely to improve performance and will even require reading the data from disk, which is less likely for a frequently used slice.

To this effect, the following statistics are kept at the level of the physical slice:

- Cumulative count of indexed random access
- Cumulative count of sequentially accessed rows
- Cumulative count of disk reads

- Cumulative count of speculative disk reads (e.g. read ahead, escalating a page read to an extent read)

- Data size

Additionally, for each logical slice, a count of random accesses where the partitioning key specifically selects this slice is kept. The access counts per logical slice are kept at the requesting host whereas the physical stats are kept by the host hosting the slice. Before a split, the host holding the slice asks all other hosts for this information.

One detects a potential need for slice motion when any host is in a state where all database buffers are being used and there is continuous reading from secondary storage. If this is coupled with CPU utilization that is less than 100% on all cores while having more threads than cores, this is a sign that slice motion is urgent.

Having first identified the host groups that need offloading, one can simply pick the slice with the most recent read IO and start moving this to a less busy server. We note that starting the motion occurs purely in the background.

We wish to avoid situations where a cluster oscillates between slice positioning schemes. For this reason, a recently moved slice should not be moved again in the near future. Rather, if such a need manifests, it is best to split the slice in question and then move one of the sides of the split to a new server. If the load had the nature of reading, this means that the access pattern is not a single page in the slice and that splitting has a chance of actually splitting a hot spot. Thus activity is an additional clue for splitting, not size alone. The access logical slice counts gathered from the requesting hosts further help dividing the slices so that both sides of the split are about equally busy.

Cost of Elasticity and Parallelism Opportunities

Excluding situations of partition motion, running an elastic cluster as opposed to a cluster with one partition per host is identical. The cost of the indirection from logical to physical slice to host group is not even visible in the execution profile.

Reading an index in index order performs a merge between partitions. With elastic cluster, the number of partitions is much greater but there is locality within each sub-stream to merge, hence the cost of merge does not go up much, besides this operation is a rarity in analytical workloads which are typically about aggregation that is not sensitive to order.

When a database function (e.g. index lookup) is shipped, the initiating server splits the request into partitions. Multiple such partitions often go to the same destination. These pre-partitioned chunks are easy to run in parallel and are guaranteed not to interfere since they touch physically distinct indices, thus there is no mutex locking or the like. Vectored index access performs a sort of the keys before fetching. This sort is in this situation partitioned, which cuts down on the sorting cost in a more than linear manner.

Thus parallelism and elastic partitioning have a synergistic interaction.

The Case for Range Partitioning

Range partitioning is generally useful when the partitioning column has application level semantics that create locality. This is typically the case for dates, where recent dates are often more frequently looked at. A scan over a range of values can thus be directed to specific partitions only whereas a hash scheme needs to ask all partitions unless the range is so small that the part of the attribute on which the hash is based is the same for the start and end of range.

In RDF, there is less locality based on attribute values since one index is approximately in insert order but other indices are generally not thus ordered. The advantage of range partitioning for RDF rather comes from the extremely uneven distribution of values for the O column. For example, common classes occur large numbers of times in the O position. With a self-tuning range partition system, one could make arbitrarily small ranges for extremely common values, leading to a more even slice size.

Range partitioning is feasible also when the partitioning key is not the leading key part, for example in the case of a POGS index. For splitting a slice of POGS under range partitioning, one takes the most space-consuming value of P and takes the middle value of O for this P and splits all according to that.

With range partitioning there is no difference between logical and physical slices and the task of finding the partition based on the partitioning key is somewhat more complex, in log time instead of linear. Still, this is a marginal impact since this can be addressed by a binary search of an array of partition boundaries.

Repartition Without Transaction Aborts

Supporting splitting and motion of slices without any global atomic operations is possible but its implementation cost is large and its benefit for an online application is marginal. For a bulk load or long-running query situation the benefit is greater. For example one could have a cloud installation automatically requisition new hardware from a cloud when starting to hit secondary storage.

This can be implemented in two ways: 1. the change is made and traffic to the old slice is redirected and repartitioned by the former host of the slice or 2. requests for a slice about to be split are held up at the requester end for any new transactions while allowing existing ones to finish before the motion/split is finalized.

The second strategy is less costly in implementation terms since it does not require requesters to deal with situations where one logical request must be treated as two streams of results. Instead, the requester must simply note that the request being formulated would hit a slice for on hold. If the transaction in question is cleared for the slice the operation proceeds, if there is a single slice in the target that is not cleared for the transaction the operation waits and retries after the partitioning information is updated. The host owning the slice notifies the requestors about which transactions are cleared for the slice. These will be the ones that already held locks in the slice when the decision to finalize the move or split was made.

This introduces new wait edges into the global transaction wait graph with the consequent possibility of deadlock but these can be handled in the usual manner. A transactional application will not be bothered by a few extra aborts and a non-transactional application like a bulk load or long read committed query will not have locks or waits in the first place.

Long running non-transactional updates, e.g. calculating entity ranks by Page rank style methods (join of everything to everything with intermediate scores written to a table) will also benefit. Such operations have a long run time and do not have natural restart points.

Limitations as of September 2011

The September 2011 prototype has not been tested with elastic partitions in duplicate, thus we have one host per host group. Schema structures are replicated but they are not elastic, since each host always has a full copy that is not further partitioned.

The operator indicates a time window for partition splitting and motion but the system itself decides what to move or split.

Availability and Download

The software deliverable can be downloaded from the following location:

ftp://download.openlinksw.com/support/lod2/virtuoso-7.0.0-cluster-alpha1.tar.gz

Please contact oerling@openlinksw.com for additional details on configuration and usage.