How does Oracle Database In-Memory Scale out?

Niloy Mukherjee, Kartik Kulkarni, Hui Jin, Jesse Kamp and Tirthankar Lahiri
Oracle Corporation, Redwood Shores, U.S.A.

Keywords: Oracle RDBMS in-Memory Option, Dual-format Distributed in-Memory Database, Scale-out, in-Memory Compression Units (IMCUs), Automated Distribution, Distributed SQL Execution.

Abstract: The Oracle RDBMS In-memory Option (DBIM), introduced in 2014, is an industry-first distributed dual format in-memory RDBMS that allows a database object to be stored in columnar format purely in-memory, simultaneously maintaining transactional consistency with the corresponding row-major format persisted in storage and accessed through in-memory database buffer cache. The in-memory columnar format is highly optimized to break performance barriers in analytic query workloads while the row format is most suitable for OLTP workloads. In this paper, we present the distributed architecture of the Oracle Database In-memory Option that enables the in-memory RDBMS to transparently scale out across a set of Oracle database server instances in an Oracle RAC cluster, both in terms of memory capacity and query processing throughput. The architecture allows complete application-transparent, extremely scalable and automated in-memory distribution of Oracle RDBMS objects across multiple instances in a cluster. It seamlessly provides distribution awareness to the Oracle SQL execution framework, ensuring completely local memory scans through affinitized fault-tolerant parallel execution within and across servers without explicit optimizer plan changes or query rewrites.

1 INTRODUCTION

Oracle Database In-memory Option (DBIM) (Oracle, 2014, Lahiri et al., 2015) is the industry-first dual format main-memory database architected to provide breakthrough performance for analytic workloads in pure OLAP as well as mixed OLTAP environments, without compromising or even improving OLTP performance by alleviating the constraints in creating and maintaining analytic indexes (Lahiri et al., 2015). The dual-format in-memory representation allows an Oracle RDBMS object (table, table partition, composite table subpartition) to be simultaneously maintained in traditional row format logged and persisted in underlying storage, as well as in column format maintained purely in-memory without additional logging. The row format is maintained as a set of on-disk pages or blocks that are accessed through an in-memory buffer cache (Bridge et al., 1997), while the columnarized format is maintained as a set of compressed in-memory granules called in-memory compression units or IMCUs (Oracle, 2014, Lahiri et al., 2015) in an in-memory column store (Lahiri et al., 2015) transactionally consistent with the row format. By building the column store into the existing row format based database engine, it is ensured that all of the rich set of features of Oracle Database (Bridge et al., 1997, Oracle, 2013) such as database recovery, disaster recovery, backup, replication, storage mirroring, and node clustering work transparently with the IM column store enabled, without any change in mid-tier and application layers.

The dual format representation is highly optimized for maximal utilization of main memory capacity. The Oracle Database buffer cache used to access the row format has been optimized over decades to achieve extremely high hit-rates even with a very small size compared to the database size. As the in-memory column store replaces analytic indexes, the buffer cache gets better utilized by actual row-organized data pages. Besides providing query performance optimized compression schemes, Oracle DBIM also allows the columnar format to be compressed using techniques suited for higher capacity utilization (Lahiri et al., 2015).

Unlike a pure in-memory database, the dual format DBIM does not require the entire database to have to fit in the in-memory column store to

Mukherjee N., Kulkarni K., Jin H., Kamp J. and Lahiri T..
How does Oracle Database In-Memory Scale out?.
DOI: 10.5220/0005497900390044
In *Proceedings of the 10th International Conference on Software Engineering and Applications* (ICSOFT-EA-2015), pages 39-44
ISBN: 978-989-758-114-4

become operational. While the row format is maintained for all database objects, the user is allowed to specify whether an individual object (Oracle RDBMS table, partition or composite subpartition) should be simultaneously maintained in the in-memory columnar format. At an object level, Oracle DBIM also allows users to specify a subset of its columns to be maintained in-memory. This allows for the highest levels of capacity utilization of the database through data storage tiering across main-memory, flash cache, solid state drives, high capacity disk drives, etc.

Section 2 presents a detailed description of the distributed architecture, with precise focus on a) complete application-transparent, extremely scalable and automated in-memory distribution mechanism of Oracle RDBMS objects across multiple instances in a cluster, and b) seamless provision of distribution awareness to the Oracle SQL execution framework.

2 DISTRIBUTED DBIM

The distributed architecture of Oracle DBIM is demonstrated through Figure 1.

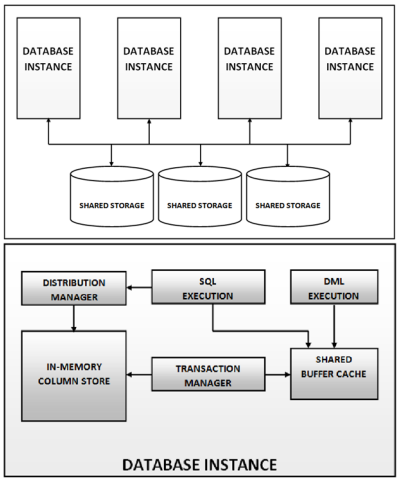

Figure 1: Distributed architecture of DBIM on Oracle RAC.

Oracle DBIM employs Oracle Real Application Cluster (RAC) (Oracle 2014) configuration for scaling out across multiple machines. RAC allows a user to configure a cluster of database server *instances* that execute Oracle RDBMS software while accessing a single database persisted in shared storage. Data of an Oracle object is persisted in traditional row major format in shared-storage as a set of extents, where each extent is a set of contiguous fixed-size on-disk pages (*Oracle Data Blocks*) as shown in Figure 2. These data blocks are accessed and modified through a shared *Database Buffer Cache*. Each individual instance can be configured with a shared-nothing *in- memory column store*. For an object that is configured to be maintained in-memory, the *distribution manager* is responsible for maintaining the corresponding *in-memory object* as a set of *In- memory Compression Units (IMCUs)* (Oracle 2014) distributed across all in-memory column stores in the cluster, with each IMCU containing data from mutually exclusive subsets of data blocks (Figure 2). Transactional consistency between an IMCU and its underlying data blocks is guaranteed by the *IM transaction manager*.

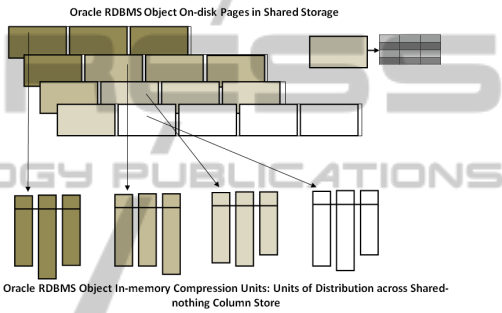

Figure 2: A 3-column Oracle RDBMS table in both row-major and in-memory columnar formats.

2.1 In-Memory Compression Unit

An In-Memory Compression Unit (IMCU) is '***populated***' by columnarizing rows from a subset of blocks of an RDBMS object and subsequently applying intelligent data transformation and compression methods on the columnarized data. The IMCU serves as the unit of distribution across the cluster as well as the unit of scan within a local node. An IMCU is a collection of contiguous in-memory extents allocated from the in-memory area, where each column is stored contiguously as a column *Compression Unit* (CU). The column vector itself is compressed with user selectable compression levels; either optimized for DMLS, or optimized for fast scan performance, or for capacity utilization (Lahiri et al., 2015).

Scans against the column store are optimized using vector processing (SIMD) instructions (Lahiri et al., 2015) which can process multiple operands in a single CPU instruction. For instance, finding the occurrences of a value in a set of values, adding successive values as part of an aggregation operation, etc., can all be vectorized down to one or

two instructions. A further reduction in the amount of data accessed is possible due to the *In-Memory Storage Indexes* (Lahiri et al., 2015) that are automatically created and maintained on each of the CUs in the IMCU. Storage Indexes allow data pruning to occur based on the filter predicates supplied in a SQL statement. If the predicate value is outside the minimum and maximum range for a CU, the scan of that CU is avoided in entirety. For equality, in-list, and some range predicates an additional level of data pruning is possible via the metadata dictionary when dictionary-based compression is used. All of these optimizations combine to provide scan rates exceeding billions of rows per second per CPU core. Apart from accelerating scans, the IM column store also provides substantial performance benefits for joins by allowing the optimizer to select Bloom filter based joins more frequently due to the massive reduction in the underlying table scan costs. A new optimizer transformation, called *Vector Group By* (Lahiri et al., 2015), is also used to compute multi-dimensional aggregates in real-time.

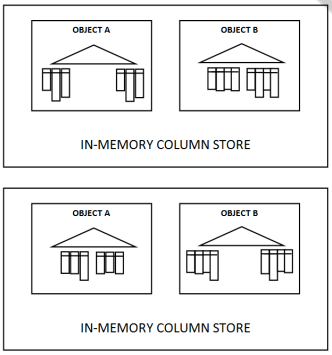

Figure 3: In-memory segments with IMCUs indexed by Oracle data block addresses (2-instance cluster).

2.2 In-Memory Column Store

The In-Memory column store is carved out from the Oracle System Global Area (SGA) (Oracle 2014) per database instance based on a size provided by the user. Logically, it is a shared-nothing container of *in-memory segments*, where each in-memory segment comprises of a set of IMCUs populated in the instance. Each in-memory segment is equipped with a data block address based *in-memory home location index* that is used for efficient lookup of an IMCU containing data from a particular underlying data block. Depending on the distribution of IMCUs, an *in-memory object* constitutes a set of one or more in-memory segments across the cluster (Figure 3).

The in-memory home location index allows for seamless integration of the in-memory column store with the traditional row-store based instance- local Oracle data access engine (Oracle 2013) that iterates over a set of row-major block address ranges. Using the same scan engine as-is allows an RDBMS object to be perpetually online for queries. For a given set of block address ranges, the access engine employs the index to detect and scan an IMCU if an IMCU covering these ranges exists locally; otherwise it falls back to the buffer cache or underlying storage. As and when IMCUs get locally populated and registered with the index, the access engine shifts from performing block based accesses to utilizing IMCUs for faster analytic query proceesing.

2.3 Distribution Manager

The primary component of the distributed architecture is the Distribution Manager. We have uniquely designed the component to provide extremely scalable, application-transparent distribution of IMCUs across a RAC cluster allowing for efficient utilization of collective memory across in-memory column stores, and seamless interaction with Oracle's SQL execution engine (Parallel 2013) ensuring affinitized high performance parallel scan execution at local memory bandwidths, without explicit optimizer plan changes or query rewrites.

The distribution manager uses a generic mechanism for population of IMCUs for a given database object. The mechanism is two-phase, comprising of a very brief centralized consensus generation phase followed by a decentralized distributed population phase. A completely centralized approach requires the coordinating instance to undergo non-trivial cross-instance communication of distribution contexts per IMCU with the rest of the instances. On the other hand, a purely de-centralized approach allows maximal scale-out of IMCU population, but the lack of consensus in a constantly changing row-store may result in a globally inconsistent distribution across the cluster.

Our two-phase mechanism aims to combine the best of both worlds. While the centralized phase generates and broadcasts a minimal distribution consensus payload, the decentralized phase allows each instance to independently populate relevant IMCUs using locally computed yet globally consistent agreements on IMCU home locations based on the broadcast consensus. In this approach, at any given time for a given object, an instance can be either a 'leader' that coordinates the consensus

gathering and broadcast, or a 'to-be follower' that waits to initiate IMCU population, or a 'follower' that coordinates the decentralized population, or 'inactive'.

The remainder of the subsection explains our approach in details. The on-disk hypothetical non-partitioned table illustrated in Figure 2 in Section 2 is used to demonstrate the various steps of the distribution mechanism in a hypothetical RAC cluster of 4 instances.

2.3.1 Centralized Phase

Distribution of a given object can be triggered simultaneously from multiple instances as and when any of the managers detects portions of an in-memory enabled object not represented by either local or remote IMCUs. Leader selection therefore becomes necessary to prevent concurrent duplicate distribution of the same object. For each object, a set of dedicated background processes per instance compete for an exclusive global object distribution lock in no-wait mode. The instance where the background process successfully acquires the object lock becomes the leader instance with respect to the distribution of that object while rest of the backgrounds bail out. Therefore, at any given time for a given object, only one instance can serve as the leader (figure 4). At this point, all other instances remain inactive as far as the given object is concerned.

The leader is responsible for gathering a set of snapshot metadata for the object, which it broadcasts to all nodes excluding itself. This snapshot provides a consensus to all nodes so that each of them can carry out IMCU population independently but through a distributed agreement. The snapshot information includes a) schema and object identifier,b) the current system change number (SCN) (Lahiri et al., 2015) of the database to get consistent snapshot of the object layout as of that SCN, c) the SCN used in the previous distribution (invalid if the object was never distributed previously), d) the set of instances participating in the population, e) the packing factor, f) the cluster incarnation, and g) the partition/subpartition number

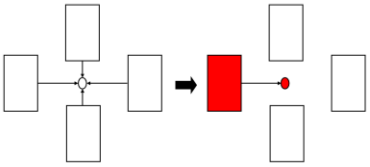

Figure 4: Election of a leader background process for an Oracle RDBMS object.

(if distribute by partition or subpartition is the option used). The extremely compact payload implies minimal inter- node communication overheads.

Once an instance receives the message from the leader, one of its dedicated background processes initiates the population task by queuing a shared request on the same object lock, changes its role of the instance from 'inactive' to 'to-be follower', and sends an acknowledgement back to the leader. However as the leader holds exclusive access on the lock, none of the instances can attain 'follower' status to proceed with the population procedure. After the leader receives acknowledgement from all instances, it downgrades its own access on the global object lock from exclusive to shared mode.

Figure 5: Consensus broadcast, acknowledgement, followed by leader downgrade.

Once the downgrade happens, the leader itself becomes a 'follower' and all 'to-be followers' get shared access on the lock to become 'followers' to independently proceed with the distributed decentralized IMCU population (Figure 5). Until all followers release access on the shared object lock, none of the instances can compete for being a leader again for the object.

2.3.2 Decentralized Population Phase

Each follower instance uses the SCN snapshot information in the broadcast message to acquire a view of the object layout metadata on-disk. Based on the previous SCN snapshot and the current one, each follower instance determines the same set of block ranges that are required to be distributed and then uses the packing factor to set up globally consistent IMCU population contexts, as demonstrated in Figure 6 (assuming a packing factor of 4 blocks). Once consistent IMCU contexts have been set up, the requirement arises to achieve distributed agreement on the assignment of instance home locations.

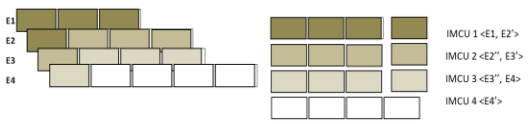

Figure 6: IMCU population context generation.

Each instance is required to independently come up with the same assignment answer for the same input key, which leads to the need for a uniform hash function. Traditional modulo based hashes may not be well suited to serve the purpose as they result in unnecessary rebalancing costs as and when cluster topology gets impacted. The distribution manager employs a technique called ***rendezvous hashing*** (Laprie, 1985) that allows each follower background process to achieve distributed agreement on the instance home location for a given IMCU. Given a key, the algorithm computes a hash weight over each instance in the set of participating instances in the payload broadcast by the leader and selects the instance that generates the highest weight as the home location.

$$f(K, N) = \sum \max(h(K, i)), i = 1..N$$

Table 1: Hypothetical home location assignments by each follower instance.

IMCU Context	IMCU Boundaries	Assignments
IMCU 1	<E1, E2'>	Inst. 1
IMCU 2	<E2'', E3'>	Inst. 2
IMCU 3	<E3'', E4'>	Inst. 3
IMCU 4	<E4''>	Inst. 4

In context of IMCU distribution, the key chosen depends on the distribution scheme. If the distribution is block range based, then the address of the first block in the IMCU context is used as the key. Otherwise, the partition number or the relative subpartition number is used as the key. As the key is chosen in consensus across all follower instances, the rendezvous hashing scheme ensures global agreement on their home locations (an example is demonstrated in Table 1). Besides achieving low computation overheads and load balancing, the primary benefit of rendezvous hashing scheme is minimal disruption on instance failure or restart, as only the IMCUs mapped to that particular instance need to be redistributed.

Once the follower background process determines the instance locations for all IMCU contexts, it divides the workload into two sets, one where IMCU contexts are assigned to its own instance and the other where they are assigned to remote instances. For the first set, it hands off the IMCU contexts to a pool of local background server processes to create IMCUs from the underlying data blocks in parallel. If an in-memory segment is not present, the population of the first local IMCU

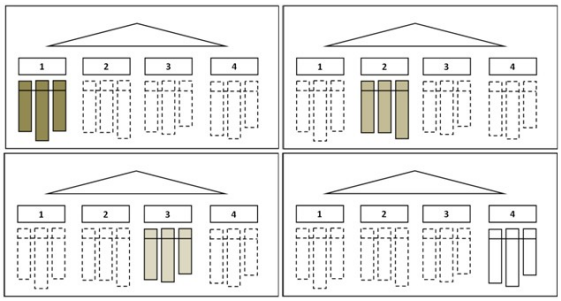

Figure 7: Logical view of in-memory home location index on completion of distribution across 4 RAC instances.

creates the in-memory segment within the column store. Once the IMCUs are created locally in the physical memory, they are registered in the block address based home location index described in section 2.2. An IMCU becomes visible to the data access as well as the transaction management components once it has been registered in the index. For the second set, the follower process iteratively registers only the remote home location metadata without undergoing actual IMCU population.

The follower background process waits for all local background processes undergoing IMCU population to complete. By the time all instances release their accesses on the global object lock, the mechanism results in laying out IMCUs consistently across all participating home location nodes resulting in a globally consistent home location index maintained locally on every instance (illustrated in Figure 7).

2.3.3 SQL Execution

The distribution manager is seamlessly integrated with the Oracle SQL execution engine (Parallel 2013) to provide in-memory distribution awareness to traditional SQL queries without explicit query rewrites or changes in execution plan. For a given query issued from any instance in the cluster, the SQL optimizer component first uses the local in-memory home location index to extrapolate the cost of full object scan across the cluster and compares the cost against index based accesses. If the access path chooses full object scan, the optimizer determines the degree of parallelism (DOP) based on the in-memory scan cost. The degree of parallelism is rounded up to a multiple of the number of active instances in the cluster. This ensures allocation of at least one parallel execution server process per instance to scan its local IMCUs.

Once parallel execution server processes have been allocated across instances, the Oracle parallel

query engine is invoked to coordinate the scan context for the given object. The query coordinator allocates (N+1) distributors, one for each specific instance 1 to N, and one that is not affined to any instance. Each distributor has one or more relevant parallel execution server processes associated with it. The coordinator acquires a consistent version of the on-disk object layout metadata to generate a set of block range based granules for parallelism. It uses the local in-memory home location index to generate granules such that their boundaries are aligned to IMCU boundaries residing within the same instances.

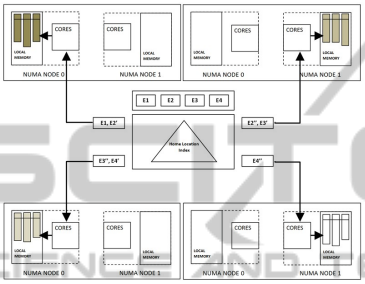

Figure 8: Home location aware parallel query execution.

The granules generated are queued up in relevant distributors based on the home location affinities. Each parallel server process dequeues a granule from its assigned distributor and hands it over to the Oracle scan engine. As described before, the scan engine uses the same in-memory index to either process IMCUs if present in the local in-memory column store, or fall back to buffer cache or disk if otherwise. Figure 8 demonstrates home location aware parallel execution of a query undergoing fully local memory scans across the cluster.

The instance alignment ensures that a granule consists of block ranges that are represented by IMCUs residing in the same local memory. IMCU boundary based alignment alleviates redundant access of the same IMCU by multiple parallel server processes. The globally consistent local home location index that the same set of granules is generated irrespective of the instance coordinating the query.

3 CONCLUSIONS

The necessity to support real-time analytics on huge data volumes combined with the rapid advancement of hardware systems has served as the 'mother of invention' of a new breed of main-memory databases meant to scale. This paper presents the distributed architecture of the Oracle Database In-memory Option. The architecture is unique among all enterprise-strength in-memory databases as it allows complete application-transparent and extremely scalable automated in-memory distribution of Oracle RDBMS objects across multiple instances in a cluster. The distributed architecture is seamlessly coupled with Oracle's SQL execution framework ensuring completely local memory scans through affinitized fault-tolerant parallel execution within and across servers, without explicit optimizer plan changes or query.

REFERENCES

Oracle Database In-Memory, an Oracle White Paper, *Oracle Openworld*, 2014.

Lahiri, T. et. al. Oracle Database In-Memory: A Dual Format In-Memory Database. *Proceedings of the ICDE* (2015).

W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, "The Oracle Universal Server Buffer Manager", in *Proceedings of VLDB '97*, pp. 590-594, 1997.

Oracle12c Concepts Release 1 (12.0.1). *Oracle. Corporation* (2013).

Parallel Execution with Oracle 12c Fundamentals, *An Oracle White Paper, Oracle Openworld, 2014.*

Laprie, J. C. (1985). "Dependable Computing and Fault Tolerance: Concepts and Terminology", Proceedings of *15th International Symposium on Fault-Tolerant Computing (FTSC-15),* pp. 2–11.

R. Greenwal, M. Bhuller, R. Stackowiak, and M. Alam, *Achieving extreme performance with Oracle Exadata*, McGraw-Hill, 2011.