Distributed Architecture of Oracle Database In-memory

Niloy Mukherjee, Shasank Chavan, Maria Colgan, Dinesh Das, Mike Gleeson, Sanket Hase, Allison Holloway, Hui Jin, Jesse Kamp, Kartik Kulkarni, Tirthankar Lahiri, Juan Loaiza, Neil Macnaughton, Vineet Marwah, Atrayee Mullick, Andy Witkowski, Jiaqi Yan, Mohamed Zait

Oracle Corporation
500 Oracle Parkway, Redwood Shores, CA 94065

{Niloy.Mukherjee}@Oracle.com

ABSTRACT

Over the last few years, the information technology industry has witnessed revolutions in multiple dimensions. Increasing ubiquitous sources of data have posed two connected challenges to data management solutions – processing unprecedented volumes of data, and providing ad-hoc real-time analysis in mainstream production data stores without compromising regular transactional workload performance. In parallel, computer hardware systems are scaling out elastically, scaling up in the number of processors and cores, and increasing main memory capacity extensively. The data processing challenges combined with the rapid advancement of hardware systems has necessitated the evolution of a new breed of main-memory databases optimized for mixed OLTAP environments and designed to scale.

The Oracle RDBMS In-memory Option (DBIM) is an industry-first distributed dual format architecture that allows a database object to be stored in columnar format in main memory highly optimized to break performance barriers in analytic query workloads, simultaneously maintaining transactional consistency with the corresponding OLTP optimized row-major format persisted in storage and accessed through database buffer cache. In this paper, we present the distributed, highly-available, and fault-tolerant architecture of the Oracle DBIM that enables the RDBMS to transparently scale out in a database cluster, both in terms of memory capacity and query processing throughput. We believe that the architecture is unique among all mainstream in-memory databases. It allows complete application-transparent, extremely scalable and automated distribution of Oracle RDBMS objects in-memory across a cluster, as well as across multiple NUMA nodes within a single server. It seamlessly provides distribution awareness to the Oracle SQL execution framework through affinitized fault-tolerant parallel execution within and across servers without explicit optimizer plan changes or query rewrites.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st – September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

1. ORACLE DBIM – AN OVERVIEW

The area of data analytics witnessed a revolution in the past decade with the deluge of data ingestion sources [1]. The past decade therefore witnessed a resurgence of columnar DBMS systems, e.g., C-Store [2] and Monet DB [3], as pure columnar format became a proven standard suited for traditional data warehousing and analytics practice where the historical data is first curated in usually dedicated data warehouses, separate from the transactional data stores used in mainstream production environment. However, being unsuitable for OLTP workloads, pure columnar format is not entirely ideal for the real-time analytics use-case model that demands high performance analysis of transactional data on the mainstream production data stores. In comparison, traditional industry-strength row-major DBMS systems [4] have been well suited for OLTP workloads but have incurred manageability and complexity overheads required in computation and maintenance of analytic indexes and OLAP engines geared towards high performance analytics [4].

Even though Oracle TimesTen [5] was one of the first industry-strength main-memory databases developed in mid 1990s, it is only over the last few years that the explosion in processing and memory capacity in commodity systems has resulted in the resurgence of main memory based database systems. These include columnar technologies such as SAP HANA [6], IBM BLU [7] etc. as well as row-oriented ones such as Oracle TimesTen and H-Store [8]. As DRAM capacity keeps on increasing and becoming cheaper [9], main memory no longer remains a limited resource. Today's multi core, multiprocessor servers provide fast communication between processor cores via main memory, taking full advantages of main memory bandwidths. Main memory is therefore being conceived by DBMS architects more as a primary storage container and less as a cache optimizing disk based accesses.

With this precursor, we present a quick overview of Oracle Database In-memory Option (DBIM) [10] [11] that was introduced in 2014 as 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 [11]. The dual-format in-memory representation (Figure 1) allows an Oracle RDBMS object (table, table partition, 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 [12], while the columnarized format is maintained as a set of compressed in-memory granules called in-memory compression units or IMCUs [10][11] in an In-memory Column Store [11] transactionally consistent with the row format [13]. By building the column store into the existing row format based database engine, it is ensured that all of the rich set of Oracle Database features [4][11] 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.

Figure 1. Dual-format representation of Oracle DBIM.

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 [11].

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 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 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.

A detailed description of Oracle DBIM features is available at the Proceedings of the 31st International Conference on Data Engineering 2015 [11]. In this paper, we primarily aim to concentrate on various aspects of the distributed architecture of Oracle DBIM, its underlying components, and related methods behind its transparency and seamlessness.

2. NEED FOR A DISTRIBUTED ARCHITECTURE

As enterprises are witnessing exponential growth in data ingestion volumes, a conventional wisdom has developed across the industry that scaling out using a cluster of commodity servers is better suited for executing analytic workloads over large data sets [13]. There are several valid reasons for development of such a perception. Scale-out enables aggregation of computational resources of multiple machines into a virtual single machine with the combined power of all its component machines allowing for easier elastic expansion [14]. Furthermore, since each node handles only a part of the entire data set, there may not be the same contention for CPU and memory resources as characterized by a centralized DBMS [15]. The scenario is particularly relevant for main-memory based RDBMSs. With increasing deluge of data volumes, the main memory of a single machine may not stay sufficient.

However, in the last couple of years or so, several researchers and industry experts have raised the question whether it is time to reconsider scale-up versus scale-out [16]. They provide evidence that the majority of analytics workloads do not process huge data sets at a given time. For example, analytics production clusters at Microsoft and Yahoo have median job input sizes less than 14 GB [17], and 90% of jobs on a Facebook cluster have input sizes under 100 GB [17]. Moreover, hardware price trends are beginning to change performance points. Today's commodity servers can affordably hold 100s of GB of DRAM and 32 cores on a quad socket motherboard with multiple high-bandwidth memory channels per socket, while high end servers such as the M6 Oracle Sun SuperCluster [18] providing up to 32 TB of DRAM and 1024 cores, are also becoming more commonplace.

As far as implementation of scale-up parallelism in main-memory architectures is concerned, it may seem less complex because the memory address space is completely shared in a single server. However, current state-of-the-art multi processor systems employ Non-uniform Memory Access or NUMA [19], a memory design where the memory access time depends on the memory location relative to the processor. Accesses from a single processor to local memory provides lower latency compared to remote memory accesses as well as alleviates interconnect contention bottlenecks across remote memory controllers. As a result, a NUMA-based distributed in-memory framework becomes necessary even in a single SMP server and gets extremely relevant for larger SMPs like the SuperCluster.

Even if a monolithic server meets the capacity/performance requirements of a data processing system, scale-out architectures can be designed to offer visible benefits of high availability and minimal recovery; features that are most relevant for a non-persistent volatile main-memory database [10]. A single main-memory database server poses the risk of a single point of failure. In addition, the recovery process (process to re-populate all data in memory) gets relatively long, leading to extended downtime. A distributed main-memory system can be designed to be fault tolerant through replication of in-memory data so that it exists at more than one site. It also provides the scope to design extremely efficient redistribution mechanisms for fast recovery.

We were motivated by these observations to design an extremely scalable high-available fault-tolerant distributed architecture within the Oracle Database In-memory Option. The remainder of the paper is organized as follows. Section 3 presents the architecture detailing the mechanisms that optimally address these observations. The uniqueness of our design is highlighted in Section 4 through a comparison study against a few relevant mainstream DBMS implementations. The features of the architecture are validated through a preliminary performance evaluation on an in-house Atomics suite, SSB workloads, and selected TPC-H benchmark queries [15].

3. DISTRIBUTED ORACLE DBIM

The distributed architecture of Oracle DBIM is demonstrated through Figure 2.

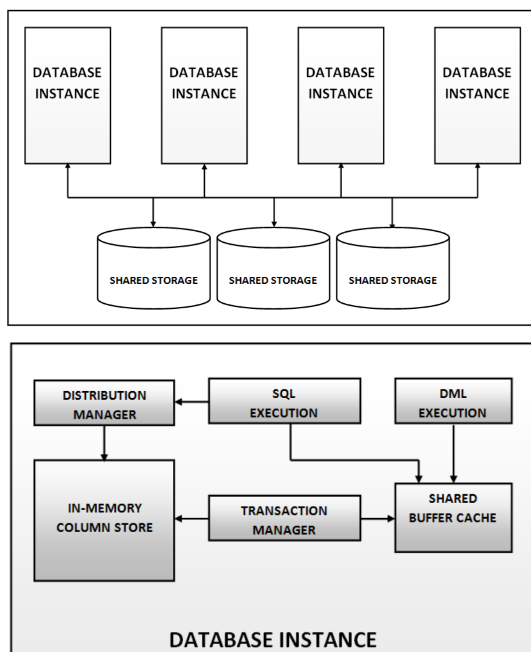

Figure 2. Distributed architecture of DBIM on Oracle RAC.

Oracle DBIM employs Oracle Real Application Cluster (RAC) [12] configurations 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. Oracle objects are 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 3. These data blocks are accessed and modified through a shared *Database Buffer Cache*. Each individual instance can also 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)* [11] distributed across all In-memory Column Stores in the cluster, with each IMCU containing data from mutually exclusive subsets of data blocks (Figure 3). Transactional consistency between an IMCU and its underlying data blocks is guaranteed by the *IM transaction manager*.

3.1 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 [11].

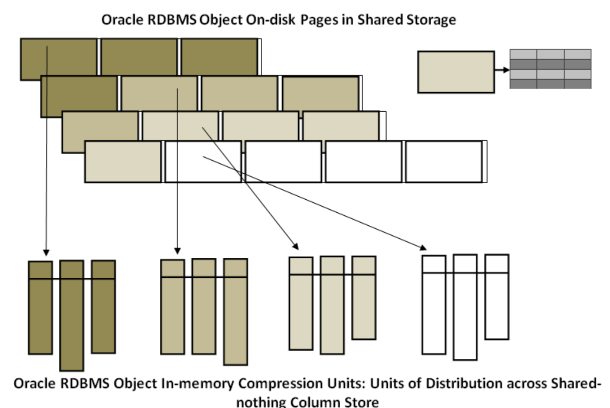

Figure 3. A 3-column Oracle RDBMS table in both row-major and in-memory columnar formats.

Scans against the column store are optimized using vector processing (SIMD) instructions [11] 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* [11] 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 entirely. 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* [11], is also used to compute multi-dimensional aggregates in real-time.

3.2 Shared Database Buffer Cache

The Oracle buffer cache is a shared collective cache of Oracle data blocks across the database cluster where each individual database server instance is configured with its own cache. The Oracle RAC cluster employs a resource control mechanism called the Global Cache Service (GCS) [12], which tracks and maintains the locations and access modes of all data blocks in the global cache. It synchronizes global cache accesses, allowing only one instance at a time to modify a data block. Read and write operations on an Oracle data block can be initiated from any of the nodes, made feasible through Cache Fusion protocol [12] that allows sharing of local buffer cache contents through fast inter-node messaging, resulting in a cluster-wide global buffer cache. The shared buffer cache component is responsible for handling all OLTP DML operations on the row-major format. Row modifications due to inserts, deletes, and updates are performed on the current version of the block transferred to the local cache. These operations employ existing Oracle data management techniques to guarantee strict ACID and robustness properties. In terms of query workloads, point queries accessing individual rows as well as OLTP index based transactional queries employ the buffer cache to access the minimal set of data blocks thereby providing least latency.

3.3 In-memory Column Store

The In-memory Column Store is carved out from the Oracle System Global Area (SGA) [12] per database instance based on a size provided by the user. If the database server has NUMA enabled [19], the underlying physical memory of the column store is allocated equally across all NUMA nodes. 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 4).

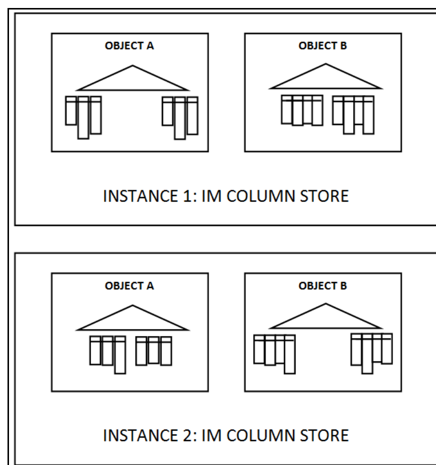

Figure 4. In-memory segments with IMCUs indexed by Oracle data block addresses (2-instance cluster)

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 [4] 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 processing.

3.4 IM Transaction Manager

Once an IMCU is populated in local memory and registered with the home location index, the IM transaction manager becomes responsible for maintaining its transactional consistency with respect to incoming DMLs. During IMCU population in a given database instance, the underlying on-disk Oracle data blocks are consistently read as of a database wide timestamp (System Change Number or SCN) [11] such that all rows are committed as of that time. Data in an IMCU is therefore a read snapshot as of its associated SCN. Each IMCU has an associated mutable metadata area called the Snapshot Management Unit (SMU) that tracks changes in rows covered by the IMCU made beyond the IMCU SCN. As mentioned in section 3.1, changes due to DMLs first modify the row-major data blocks through the buffer cache. Once a transaction commits, the changes are broadcast to be journaled in the relevant set of SMUs across the cluster. When a subsequent scan runs against the IMCU, it fetches the changed column values for all the rows that have been modified within the IMCU at an SCN earlier than the SCN of the scan As changes accumulate in the SMU, retrieving data from the transaction journal causes increased overhead for scans. Therefore, a background ***repopulate*** mechanism is periodically invoked to rebuild the local IMCU at a new SCN.

3.5 Distribution Manager

The primary component of the distributed architecture is the Distribution Manager. We have uniquely designed the component to provide the following set of capabilities; a) extremely scalable application-transparent distribution of IMCUs across a RAC cluster allowing for efficient utilization of collective memory across in-memory column stores, b) high availability of IMCUs across the cluster guaranteeing in-memory fault-tolerance for queries, c) application-transparent distribution of IMCUs across NUMA nodes within a single server to improve vertical scale-up performance, d)efficient recovery against instance failures by guaranteeing minimal rebalancing of IMCUs on cluster topology changes, and e) seamless interaction with Oracle's SQL execution engine [20] ensuring affinitized high performance parallel scan execution at local memory bandwidths, without explicit optimizer plan changes or query rewrites.

3.5.1 Distribution Schemes

By default, the architecture provides fully automated application-transparent and data independent distribution of IMCUs for a given object. However, it allows a user to explicitly specify a distribution scheme per RDBMS object to control the layout of its corresponding in-memory object. If no such scheme is specified, the distribution manager automatically chooses the best scheme

that would provide maximal query load balancing and throughput scale out.

1. DISTRIBUTE BY PARTITION: Applicable for a partitioned table, this scheme assigns all IMCUs from the same partition to the same instance and distributes different partitions to different instances. This type of distribution is specially suited to hash partitions which usually exhibit uniform data distribution pattern across partitions.

2. DISTRIBUTE BY SUBPARTITION: For a composite partitioned table, distributing by sub-partition allows the IMCUs of the sub-partitions sharing the same sub-partition criteria to be co-located in the same instance. This scheme can be helpful when the top-level partitioning criteria could cause skewed data access. A composite partitioned table may also be distributed by partition where subpartitions of the same partition are collocated in a database instance.

3. DISTRIBUTE BY ROWID/BLOCK RANGE: When a table is not partitioned, or the partitioning criteria used would cause severely skewed access, the user can specify this option to ensure that IMCUs of an in-memory object are distributed across the multiple instances in the cluster. IMCUs within the same object are assigned home locations independently of each other while maintaining uniformity across the cluster.

4. AUTO DISTRIBUTION: Choosing this scheme (which is the default distribution mode) indicates that it is up to the system to automatically select the best distribution scheme from the above three, based on the table's (non, sub) partitioning criteria and optimizer statistics. For example, Figure 5 demonstrates the choice of distribution scheme based on the cardinality of the par of a composite partitioned table to attain maximal scale-out.

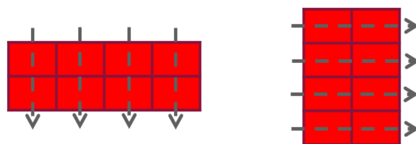

Figure 5. A 2x4 composite partitioned table is distributed by subpartition while a 4x2 one is distributed by partition.

3.5.2 *Distribution Mechanism*

Irrespective of the distribution scheme employed, 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.

The 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 3 in Section 3.1 is used to demonstrate the various steps of the distribution mechanism in a hypothetical RAC cluster of 4 instances, each instance running on a 2-socket NUMA enabled server.

3.5.2.1 *Centralized Coordination 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 6). At this point, all other instances remain inactive as far as the given object is concerned. The global object distribution lock is a purely non-blocking lock. It does not block concurrent DMLs, concurrent queries, as well as concurrent IMCU repopulation operations on the same object.

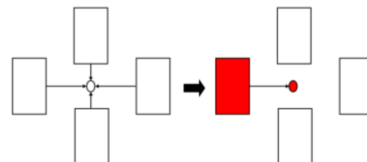

Figure 6. Election of a leader background process

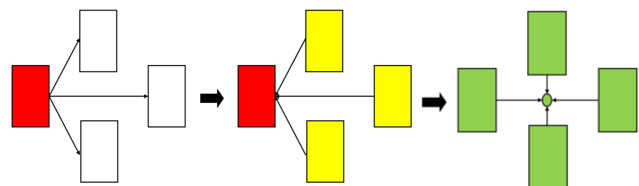

Figure 7. Consensus broadcast, acknowledgement, followed by leader downgrade

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.

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 7). Until all followers release access on the shared object lock, none of the instances can compete for being a leader again for the object.

3.5.2.2 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 8 (assuming a packing factor of 4 blocks)

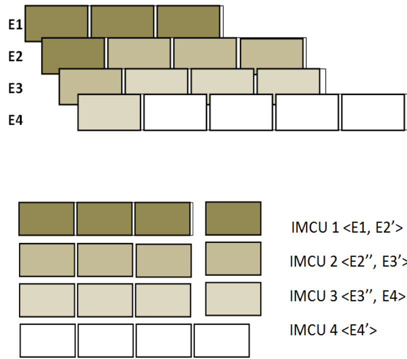

Figure 8. IMCU population context generation.

Once consistent IMCU contexts have been set up, the requirement arises to achieve distributed agreement on the assignment of instance home locations. 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*** [21] 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.

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

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 (explained in section 3.4.4).

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

IMCU Context	IMCU Boundaries	Assignments
IMCU 1	<E1, E2'>	Inst. 1, NUMA 0
IMCU 2	<E2'', E3'>	Inst. 2, NUMA 1
IMCU 3	<E3'', E4'>	Inst. 3, NUMA 0
IMCU 4	<E4''>	Inst. 4, NUMA 1

Unlike a cluster where topology changes are inevitable, NUMA topology within a single server remains static over the lifetime of the server. Therefore, more traditional modulo based hash functions can be employed to determine NUMA node locations for individual IMCUs. The hash keys used are the same as the ones used for determining the instance home locations.

Once the follower background process determines the instance and the NUMA 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 creates the in-memory segment within the column store. Once the IMCUs are created locally in the physical memory of the assigned NUMA node, they are registered in the block address based home location index described in section 3.2. 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.

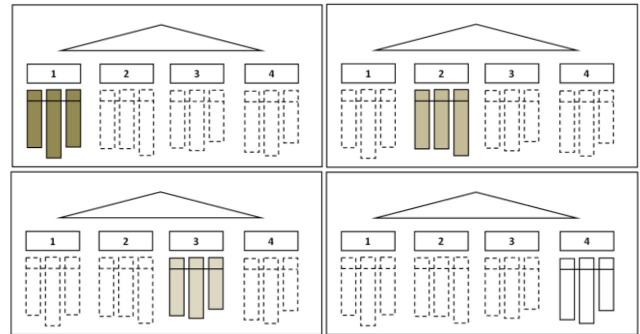

Figure 9. Logical view of in-memory home location indexes on completion of distribution across 4 RAC instances.

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 Figures 9 and 10).

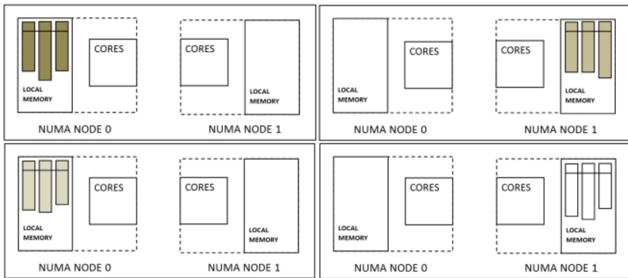

Figure 10. Physical view of in-memory column stores on completion of distribution.

3.5.3 Redistribution

The redistribution process is triggered per object when the distribution manager detects a change in cluster topology. Since redistribution results in inevitable data movement, the mechanism has been designed to provide as minimal data movement as possible.

The redistribution mechanism mostly follows the same principles as the distribution one except a few additional steps in the decentralized distribution phase. Rather than employing a new snapshot, the leader process reuses the snapshot of the most recent distribute such that the consensus on the IMCU contexts stays the same. The follower background process determines home locations for the same IMCU contexts based on the new set of instances and compares them with the original ones from the home location index. If the locations match, no further operations are required. If both locations are remote, only the metadata gets updated in the index. If the new location is local and the old location is remote, the IMCU context becomes a candidate for population. If the new location is remote and the old location is local, the IMCU context is a candidate for drop. This results in two sets of contexts, one that is required to be populated while the other that is required to be dropped. The follower waits till all IMCUs in the first set are populated in parallel. On completion of the population, it drops all IMCUs in the second set.

The above scheme ensures highest availability and minimal rebalancing of IMCUs during the redistribution process. On an instance failure, the rendezvous hashing scheme generates new locations for IMCU contexts that were affined to the failed instance. Once it becomes active again, the home locations are reverted back for these IMCUs only (Figure 11).

3.5.4 Availability Options

By default, the distribution manager ensures that a range of data blocks for an object gets represented by a single IMCU across the cluster. However, Oracle DBIM option also allows users to specify 1-safe as well as (N-1)-safe IMCU redundancy options [22] for a table, partition, or subpartition.

The decentralized distribution mechanism seamlessly provides the specific availability option when configured. For the 1-safe scenario, the rendezvous hashing based scheme is used to additionally select the instance that generates the lowest hash weight for a given key as the distributed agreement for secondary home location for an IMCU. A follower instance populates an IMCU locally if either the primary or the secondary instance assignments match its own identity. The home location index leaf nodes keep track of both primary and secondary home locations. For the (N-1)-safe scenario, each follower instance populates all IMCUs. While there is no requirement for distributed agreement in this scenario, the IMCU context consensus is still required to keep all local home location indexes in sync.

Figure 11. Rebalancing on failure and restart of instance 2.

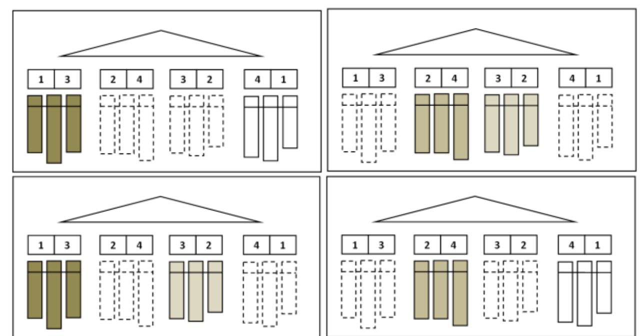

Figure 12. Logical view of in-memory home location index on completion of 1-safe distribution across 4 RAC instances.

Figure 12 illustrates the layout of IMCUs after distribution for the 1-safe scenario. The set of availability options are optimally suited for the requirements of star-schema based analytic workloads. For very large fact tables, no redundancy or 1-safe redundancy allows for efficient utilization of collective memory. For small tables containing just a few IMCUs (such as small dimension tables that

frequently participate in joins), it is advantageous to have (N-1)-safe redundancy on every instance, in order to ensure that all queries obtain local access to them at all times.

3.6 Distributed SQL Execution

The distribution manager provides IMCU NUMA and instance 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 a full object scan across the cluster and compares it against the cost of an 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 DOP 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. If the determined DOP is greater than the sum of all NUMA nodes in the cluster, the parallel execution server processes are distributed equally across all NUMA nodes so that these processes can advantages of fully local memory accesses within a server without bottlenecking remote memory controllers.

Once parallel execution server processes have been allocated across instances and NUMA nodes, 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 instance affined distributor has multiple sub-distributors allocated for each NUMA node. Each (sub) 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 instance and NUMA nodes.

The granules generated are queued up in relevant (sub) distributors based on the home location affinities. Each parallel server process dequeues a granule from its assigned (sub) 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 13 demonstrates home location aware parallel execution of a query undergoing fully local memory scans across the cluster.

The instance and NUMA 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.

3.6.1 In-memory Fault Tolerance

Queries on Oracle objects that are distributed without redundancy incur buffer cache or storage I/O when a set of IMCUs becomes unavailable due to instance failure until they are redistributed across active instances. However queries on objects distributed with 1-safe redundancy are in-memory fault tolerant under a single instance failure within the cluster. Similarly, queries on objects distributed with (N-1)-safe redundancy are in-memory fault tolerant under (N-1) instance failure within the cluster. For tables that are distributed with 1-safe redundancy, the query coordinator first detects and caches the status of each instance with respect to the snapshot of the most recent in-memory distribution. The query coordinator decides to employ either the complete set of primary home locations or secondary home locations (based on a hash function) during the granule generation process. This ensures consistent execution scale-out and minimal skew within a single query as well load balancing across multiple queries on the same object when all instances are active. However, if the selected location for an IMCU is inactive, the active instance gets chosen.

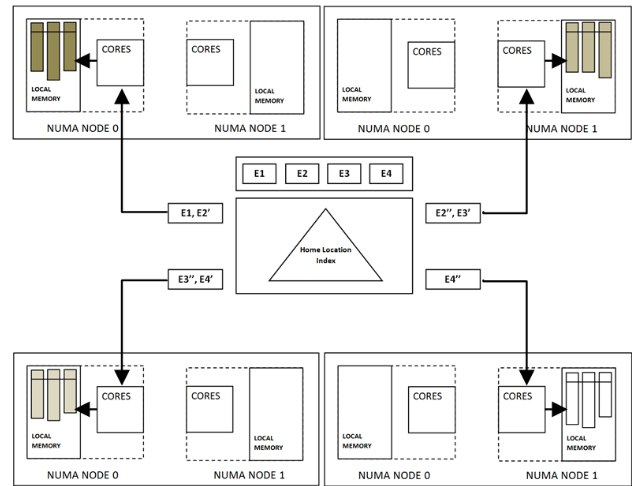

Figure 13. Home location aware parallel query execution.

4. UNIQUENESS OF ARCHITECTURE

In this section, we aim to highlight the uniqueness of our architecture through a brief comparative study against relevant enterprise-strength main-memory databases, namely SAP HANA [6] and IBM DB2 with BLU [7], by contrasting architectural approaches taken in the areas of distribution, scalability, availability and recovery.

HANA implements its distributed architecture through the use of multiple purely share-nothing INDEX servers in one SAP HANA cluster [6], where each INDEX server hosts an explicitly user-determined partition of the database. The association of an INDEX server with its underlying components (tables or partitions) is maintained in a single MASTER NAME server in steady state. As a result, unless the application itself is partition aware (no application transparency), SQL execution has to always incur a multi-hop process, in which the MASTER Name Server needs to do a lookup and forward incoming connections to the right Index Server hosting the corresponding table. Besides visible manageability overheads in configuring a cluster with various explicit roles or in defining explicit database partitions, the bottleneck of having a single NAME server results in a considerable increase in access latency.. The absence of intra-table or intra-partition distribution leads to poor load balancing and throughput scale-out, especially when a single table or

partition is accessed heavily. In contrast, Oracle DBIM allows for fully application transparent load balanced intra-table and intra-partition distribution without incurring additional data manageability overheads. SQL execution requests can be forwarded to any instance for coordination as in-memory location information is available on all instances, alleviating the need to configure a cluster explicitly with separate classes of servers.

As far as in-memory availability is concerned, HANA does not offer any redundancy options. As a result, even a single index server failure implies complete loss of in-memory data for a particular database partition. Until the entire database partition is re-populated in a standby index server, the components in this database partition remain offline for applications. Even during node addition, the entire database has to be quiesced and a full backup has to be taken. Since component locations are reassigned in a round robin manner, addition of even a single node results in unnecessary movement of almost all components. This results in significant availability overheads and superlative amounts of application downtime. In contrast, Oracle DBIM provides multiple redundancy options of the in-memory data as a result of which an application simply reconnects to any of the surviving instances and continues executing in-memory on instance failures. Even with no redundancy, only a percentage of the in-memory object is lost and the underlying Oracle RAC framework allows access of data corresponding to the lost IMCUs from the row-store through the shared buffer cache while the data gets redistributed. The rendezvous scheme based IMCU-by-IMCU distribution ensures minimal data movement during redistribution. Overall, these approaches ensure maximal in-memory utilization for SQL execution under cluster topology changes.

IBM DB2 with BLU allows columnar and in-memory representation of the on-disk tables via column groups, pages and extents. The in-memory columnar representation is a per-node feature and does not span across multiple nodes in a cluster. Such a distribution approach does not address either load or throughput scale out. On node failures, the in-memory presence for the on-disk persistent tables is lost. Since the in-memory column-groups are not distributed or replicated across the other nodes in a cluster, they become unavailable. While the actual data on the on-disk tables is itself available on any single node failure, the in-memory performance outage until node-restoration and in-memory repopulation is unavoidable. In contrast, collective memory utilization, redundancy and consistent rendezvous hashing enable Oracle DBIM on RAC to scale out and hence make the in-memory performance highly available in spite of node failures.

The study clearly reveals that while enterprise-strength main-memory databases are relying on high-speed DRAM technology, columnar format based optimizations, and vertical scale up based processing to achieve obvious performance gains in workloads, only a few focus on distributed scale out and availability. This is very much in contrast with the approaches taken by a different ecosystem of emerging databases under the category of 'NoSQL' databases [23] that focus purely on high-available distributed scale out of memory or storage capacity and access throughput. They claim to scale beyond the levels of traditional databases because they compromise either/both consistency and atomicity among the traditional ACID property set. These solutions specialize in simple operations such as single lookups, small updates, etc., and are not equipped to handle complex query workloads such as multi-node joins, complex transformations, and other rich set of data management features provided by enterprise-strength databases. The distributed architecture of Oracle DBIM, in addition to scaling up mixed OLTAP workloads, provides a complete scale out solution with collective memory utilization, redundancy, availability, and efficient failure handling by redistribution. The unique architecture therefore brings the best of both worlds - distributed scale out focus of the emerging NoSQL approaches and completeness of the main-memory Oracle RDBMS.

5. PRELIMINARY EVALUATION

From its release in 2014 onwards, Oracle DBIM option has been evaluated exhaustively through several real-world enterprise workloads. In this paper, however, we present a preliminary evaluation primarily to validate the performance, scalability, and availability aspects of the distributed architecture through a set of experiments. The experiments have been designed to demonstrate and verify the capabilities of the architecture that include a) scale-out of IMCU distribution throughput in RAC, b) in-memory aware distributed SQL execution scale-out, c) impact of in-memory distribution awareness in cluster-wide SQL execution, d) SQL execution fault-tolerance in RAC, and e) IMCU NUMA-aware scale-up within a single server.

5.1 Hardware Setup

The experiments are conducted on Oracle Exadata Database Machine [22], a state-of-the-art database SMP server and storage cluster system introduced in 2013. The NUMA experiment is conducted on an X4-8 single node machine equipped with 8 15-core Intel Xeon processors and 2TB DRAM. The rest of the experiments are conducted on an X4-2 RAC configuration comprising up to 8 database server nodes, each equipped with 2 12-core Intel Xeon processors and 256GB DRAM, and 14 shared storage servers amounting to 200TB total storage capacity, over a state-of-the-art Direct-to-Wire 2 x 36 port QDR (40 Gb/sec) InfiniBand interconnect.

5.2 Distribution Experiments

The objective of this set of experiments is to verify whether IMCU distribution throughput scales out with the number of database server instances in the RAC cluster. A set of two experiments are performed; one using a non-partitioned table and the other using a range-hash composite partitioned table.

5.2.1 Non-partitioned Table Distribution

A 13-column 1-billion row non-partitioned 'atomics' table with storage size of 64GB is chosen for this experiment. The table is configured with default compression and auto distribution parameters. A 2x compression ratio is achieved with the default compression level resulting in the creation of 16,640 IMCUs. The size of the table is kept constant while the number of nodes in the cluster is varied between 1, 2, 4, and 8 respectively. Figure 16 demonstrates almost linear scale-out in throughput of distribution of 16,640 IMCUs with increasing number of instances.

5.2.2 Composite-partitioned Table Distribution

The TPC-H lineitem schema is chosen for this experiment. The lineitem table is 84-way partitioned on (l_shipdate), with each partition 256-way hash partitioned on (l_partkey). The table is configured with default compression and distribution parameters. The on-disk size of the table is varied between 128 GB, 256GB, 512GB, and 1TB as the number of instances varies between 1, 2,

4 and 8. Figure 14 demonstrates nearly linear scale-out in distribution throughput of the lineitem table with the scale out in capacity.

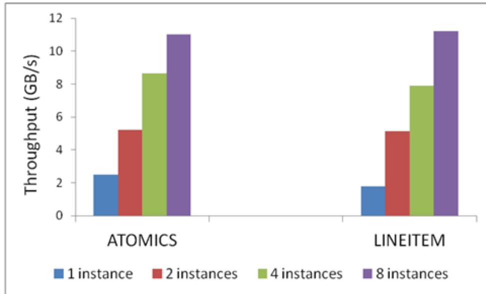

Figure 14. Distribution throughput scale-out.

5.3 Distributed Query Execution

A set of three experiments is performed on the 13-column 64GB ‘atomics’ table to observe and validate whether distribution aware parallel query execution scales out with the number of database instances in the cluster. The table is auto-distributed based on block ranges without redundancy. The size is kept constant while the number of instances is varied from 1, 2, 4, and 8.

```
Query Set 1:

select count(*) from atomics where -1.0*(uniq100m-0.5)/(0.5-uniq100m)*(rand15-0.5)/(0.5-rand15)*(rand255-0.5)/(0.5-rand255) < 99999999;

select count(*) from atomics where (uniq100m-0.5)/(0.5-uniq100m)*(rand15-0.5)/(0.5-rand15)*(rand255-0.5)/(0.5-rand255)*(rand4095-0.5)/(0.5-rand4095) < 99999999;

select count(*) from atomics where -1.0*(uniq100m-0.5)/(0.5-uniq100m)*(rand15-0.5)/(0.5-rand15)*(rand255-0.5)/(0.5-rand255)*(rand4095-0.5)/(0.5-rand4095)*(rand64k-0.5)/(0.5-rand64k) < 99999999;

Query Set 2:

select max(-2.0*(uniq100m-0.5)/(0.5-uniq100m)*(rand15-0.5)/(0.5-rand15)*(rand255-0.5)/(0.5-rand255)) from atomics where uniq100m < 99999999;

select max ((uniq100m-0.5)/(0.5-uniq100m)*(rand15-0.5)/(0.5-rand15)*(rand255-0.5)/(0.5-rand255)*(rand4095-0.5)/(0.5-rand4095)) from atomics where uniq100m < 99999999;

select max (-2.0*(uniq100m-0.5)/(0.5-uniq100m)*(rand15-0.5)/(0.5-rand15)*(rand255-0.5)/(0.5-rand255)*(rand4095-0.5)/(0.5-rand4095)*(rand64k-0.5)/(0.5-rand64k)) from atomics where uniq100m < 99999999;

Query Set 3:

select max (uniq100m) from atomics where randstringsize26 like '%imkee%';
select max (uniq100m) from atomics where randstringsize26 like '%indir%';
select max (uniq100m) from atomics where randstringsize26 like '%abcde%';

Query Set 4:

select max (rand15) from atomics where uniq100m <= 25600;
select max (rand15) from atomics where uniq100m <= 409600;
select max (rand15) from atomics where uniq100m <= 6553600;
```

Figure 15. Queries used for throughput scale-out experiment

Four different query sets are selected for each of these experiments (shown in Figure 15). Query set 1 constitutes three queries on the table with increasing complexity of where clause predicates. Query set 2 comprises of three queries with increasing complexity in the select clause. Query set 3 comprises of three queries with different ‘like’ predicates. Query set 4 comprises of three simple queries with a single ‘<=’ predicate with increasing selectivity percentage. The column ‘uniq100m’ is a number column with unique values from 1 to 1 billion. The column ‘randstringsize26’ consists of uniform random strings derived from alphabets ‘a,b,c,..,m’. The results (Figure 16) demonstrate near-linear scale-out on sets 1, 2, and 3 where the queries are CPU-bound. Queries in set 4 exercise in-memory storage indexes to prune a very large percentage of IMCUs irrelevant to the predicates. As a result, these queries are neither CPU nor memory bound. Therefore, throughput of such queries is not expected to scale with the number of instances. Moreover, For such queries, cross-instance messaging and execution overheads of setting up parallel server processes may dominate elapsed times when compared to single instance runs. The results however demonstrate that these overheads do not regress with increasing number of database instances in the cluster.

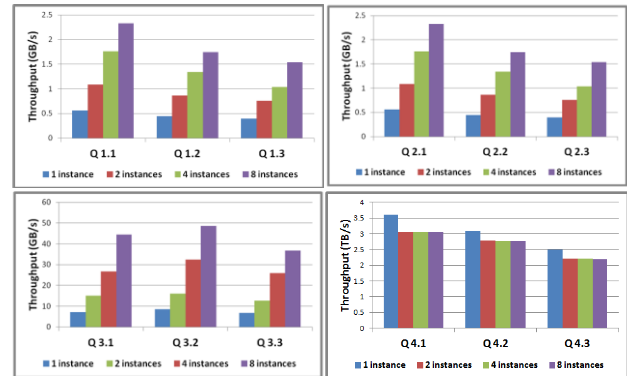

Figure 16. Distributed execution throughput scale-out

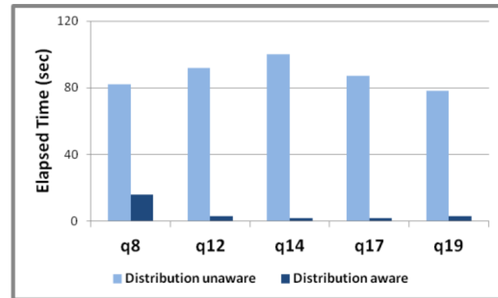

Figure 17. Elapsed time improvements with distribution-awareness in TPC-H query executions

5.4 In-memory Distribution Awareness

This experiment is conducted to observe and validate the impact of in-memory distribution awareness in execution of cluster-wide analytic query performance. The 1TB scale factor TPC-H schema is employed for the experiment. The ‘lineitem’ and ‘orders’ fact tables are 84x256-ways range-hash partitioned while the rest of the dimension tables are 256-way hash partitioned. All tables are auto distributed with no redundancy across 8 instances. A subset of 5 randomly chosen TPC-H queries (q8, q12, q14, q17, and q19) is selected for this experiment. All five queries are executed with a DOP of 256. Figure 17 demonstrates performance gains in orders of 20x to 40x over executions with distribution awareness disabled in parallel query granule generation phase, thereby validating the impact of in-memory home location awareness in cluster-wide query execution.

5.5 In-memory Fault Tolerance

The objective of this experiment is to validate in-memory fault tolerance of distributed query execution under 1-safe redundancy. The same 1TB scale factor TPC-H tables are employed for the experiment, but they are distributed with 1-safe redundancy across 8 instances. The same subset of 5 TPC-H queries (q8, q12, q14, q17, and q19) is used for this experiment. These queries are first executed across all 8 instances with a DOP of 256 and their elapsed times noted. Subsequently, one of the instances is aborted manually and the same queries are executed from one of the remaining 7 instances. It is observed from the performance graph (Figure 18) that a single instance failure has no visible effect on the elapsed times of the queries.

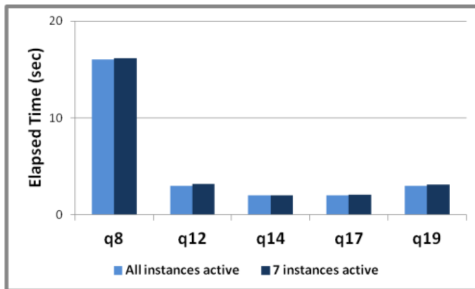

Figure 18. Query elapsed times with 1-safe redundant distribution on a single instance failure

5.6 NUMA Aware Query Execution

The objective of the experiment is to observe whether IMCU NUMA-affined query execution yields better throughput compared to the same in-memory execution but without NUMA awareness. The 1TB SSB schema is used for this experiment. The tables are distributed in-memory in a single X4-8 database instance across 8 NUMA nodes. 3 sets of 3 SSB queries with DOP of 128 involving joins between fact and multiple dimension tables are executed with and without IMCU NUMA location awareness in parallel query granule generation phase. Even though all 18 executions are driven using the column store, the results from Figure 19 demonstrate 150-250% improvements in query elapsed times when execution is IMCU NUMA-aware.

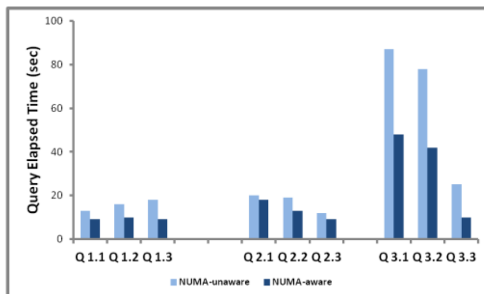

Figure 19. Elapsed time improvement with NUMA awareness

5.7 Evaluation Summary

The preliminary evaluation provides a compact yet complete demonstration of the scalability and availability capabilities of Oracle DBIM. The results do demonstrate that Oracle DBIM scales out seamlessly across a RAC cluster, in terms of a) memory capacity, b) in-memory distribution throughput, and c) distributed query execution throughput. As far as vertical scale-up is concerned, the results demonstrate the relevance of NUMA based distribution and NUMA aware SQL execution to improve query throughputs even within a single machine database. These performance gains have the potential to attain further significance as future generation of processor architectures continue optimizing local memory accesses over remote ones [19].

6. CONCLUSION

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 optimized for mixed OLTAP environments. Oracle introduced the Database In-memory Option (DBIM) in 2014 as the industry-first dual format in-memory RDBMS highly optimized to break performance barriers in analytic query workloads without compromising or even improving performance of regular transactional workloads. As enterprises continue witnessing exponential growth in data ingestion volumes, the ability to scale elastically becomes an important design requirement for state-of-the-art data management architectures. This paper presents the high-available fault-tolerant 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, as well as across multiple NUMA nodes within a single server. 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 rewrites.

7. ACKNOWLEDGMENT

We acknowledge the contributions of all members in Oracle SQL, Data, Space, Transactions, Functional Testing and Stress Testing teams involved in the entire product lifecycle, from brainstorming to product design, product development, and quality assurance. We also thank Niraj Srivastava, Linan Jiang, and Indira Patil for helping us with performance evaluation of the architecture. Last but not the least; we acknowledge the vision and passion of Amit Ganesh towards making Oracle DBIM a reality.

8. REFERENCES

[1] Elmqvist, N., Irani, P. Ubiquitous Analytics: Interacting with Big Data Anywhere, Anytime. *Computer*, 46, 4 (April 2013),

[2] Stonebraker, M., Abadi, D., et. al. C-Store: A Column-oriented DBMS. *Proceedings of the 31st VLDB Conference* (2005)

[3] Boncz, P., A., Grust, T. et. al. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2006)

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

[5] Lahiri, T., Neimat, M. and Folkman, S. Oracle TimesTen:. *IEEE Data Eng. Bull.* 36, 2 (2013), 6-13

[6] Färber, F., May, N., Lehner, W., Große, P., Müller, I., Rauhe, H. and Dees, J. The SAP HANA Database -- An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28-33.

[7] Raman, V. et. al. DB2 with BLU acceleration: so much more than just a column store, Proceedings of the VLDB Endowment, 6, 11 (2013), 1080-1091

[8] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. et. al. H-Store: A High-Performance, Distributed Main memory Transaction Processing System. *Proceedings of the VLDB Endowment.* 1, 2 (2008), 1496-1499

[9] P.A. Boncz, S. Manegold, and M.L. Kersten, "Database Architecture Optimized for the New Bottleneck: Memory Access", in *Proceedings of VLDB '99*, pp. 54-65. 1999.

[10] Oracle Database In-Memory, an Oracle White Paper, *Oracle Openworld*, 2014

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

[12] 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.

[13] Apache Hadoop. http://hadoop.apache. org/. Accessed: 08/09/2011

[14] J. Dean and S. Ghemawat. "MapReduce: Simpli- fied Data Processing on Large Clusters". OSDI. 2004.

[15] Stefan Hildenbrand, Scaling Out Column Stores: Data, Queries, and Transactions, ETH Zürich, Diss. Nr. 20314, 2012.

[16] M. Michael, "Scale-up x Scale-out: A Case Study using Nutch/Lucene". Proceedings of the IEEE International Symposium on Parallel and Distributed Processing. IPDPS'07. IEEE, 2007, pp. 1–8.

[17] Raja Appuswamy et. al., Scale-up vs scale-out for Hadoop: time to rethink?, Proceedings of the 4th annual Symposium on Cloud Computing, October 01-03, 2013, Santa Clara, California

[18] Oracle America, "Oracle SuperCluster M6-32: Taking Oracle Engineered Systems to the Next Level", An Oracle Whitepaper, Sept 2013.

[19] Sergey Blagodurov; (2011-05-02). "A Case for NUMA-aware Contention Management on Multicore Systems" (PDF). Simon Fraser University. Retrieved 2014-01-27.

[20] Parallel Execution with Oracle 12c Fundamentals, An Oracle White Paper, Oracle Openworld, 2014

[21] 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

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

[23] Strauch, Ch. 2011. NoSQL Databases. Lecture Selected Topics on Software-Technology Ultra-Large Scale Sites, Manuscript. Stuttgart Media University, 2011, 149 p., http://www.christof-strauch.de/nosqldbs.pdf