Oracle Database In-Memory: A Dual Format In-Memory Database

Tirthankar Lahiri[#1], Shasank Chavan[#2], Maria Colgan[#3], Dinesh Das[#4], Amit Ganesh[#5], Mike Gleeson[#6], Sanket Hase[#7], Allison Holloway[#8], Jesse Kamp[#9], Teck-Hua Lee[#10], Juan Loaiza[#11], Neil Macnaughton[#12], Vineet Marwah[#13], Niloy Mukherjee[#14], Atrayee Mullick[#15], Sujatha Muthulingam[#16], Vivekanandhan Raja[#17], Marty Roth[#18], Ekrem Soylemez[#19], Mohamed Zait[#20]

[#] *Server Technologies, Oracle America*

400 Oracle Parkway, Redwood Shores, CA 94065, USA

{[1] tirthankar.lahiri, [2]shasank.chavan, [3]maria.colgan, [4]dinesh.das, [5]amit.ganesh, [6]mike.gleeson, [7]sanket.s.hase, [8]allison.holloway, [9]jesse.kamp, [10]teck.hua.lee, [11]juan.loaiza, [12]neil.macnaughton, [13]vineet.marwah, [14]niloy.mukherjee, [15]atrayee.mullick, [16]sujatha.muthulingam, [17]vivekanandhan.raja, [18]martin.roth, [19]ekrem.soylemez, [20]mohamed.zait}@oracle.com

Abstract—**The Oracle Database In-Memory Option allows Oracle to function as the industry-first dual-format in-memory database. Row formats are ideal for OLTP workloads which typically use indexes to limit their data access to a small set of rows, while column formats are better suited for Analytic operations which typically examine a small number of columns from a large number of rows. Since no single data format is ideal for all types of workloads, our approach was to allow data to be simultaneously maintained in both formats with strict transactional consistency between them.**

The new columnar format is a pure in-memory format with no impact to the on-disk representation. Tables required for fast analytics can be populated into the In-Memory column store. In-Memory columnar formats allow a variety of optimizations including various levels of compression, SIMD vector processing, and in-memory storage indexes. The Oracle in-memory column format thus results in per CPU core scan speeds exceeding many billions of rows per second. Furthermore, the greatly accelerated scan speed enables a variety of query optimizations such as In-Memory aggregation (Vector Group By), a real-time computation of cube aggregations that converts costly joins and aggregations into a series of filtered scans. The in-memory column store can be scaled out on a RAC cluster with additional high availability via in-memory duplication.

OLTP updates and highly selective lookups on the same tables can be performed via the existing in-memory row store, i.e. the buffer cache. This allows the DBA to drop indexes required purely for analytics, and use the column store instead. Dropping analytic indexes may provide a substantial speedup for OLTP DML operations by eliminating costly index maintenance.

The in-memory column store is seamlessly built in to the Oracle Database engine, therefore ensuring that all of the rich functionality and High Availability mechanisms of the Oracle Database work transparently with Database In-Memory.

I. Introduction

Commodity hardware systems now have memory sizes in the hundreds of gigabytes, while terabyte range memory sizes such as with the M6 SuperCluster [1], which has up to 32 TB of DRAM, are becoming more commonplace. It is no surprise therefore, that In-Memory Database technology is becoming mainstream for enterprise applications (e.g. TimesTen [2]). The key novelties in the approach taken with the Oracle Database In-Memory feature are as follows:

Dual format: It is well known that columnar data format is well suited for analytics since analytic queries typically access a small number of columns in a large number of rows. This is why columnar storage is adopted by analytics DBMS's like MonetDB [4] and C-Store [5]. On the other hand, row-storage is better for OLTP workloads that typically access a large number of columns in a small number of rows. This is why row-storage is used by most OLTP databases, including present-generation in-memory OLTP databases such as TimesTen [2] and H-store [3].

Since neither format is optimal for all workloads, Oracle Database In-Memory features a *Dual Format* in-memory representation (Figure 1) with data simultaneously accessible in an In-Memory column store (IM column store) as well as in an in-memory row store (the buffer cache). This approach is somewhat similar to the dual representation proposed in [9].

Note that this dual format representation does not double memory requirements. The Oracle Database buffer cache [6] has been highly optimized over decades, and achieves extremely high hit-rates even with a very small size compared to the database size. Also, since the IM column store can replace analytic indexes, the buffer cache size can be further reduced. We therefore expect customers to be able to use 80% or more of their available memory for the IM column store.

Figure 1: Dual-Format Database Architecture

Unlimited Capacity: The entire database is not required to have to fit in the IM column store since it is pointless to waste memory storing data that is infrequently accessed. It is possible to selectively designate important tables or partitions (for instance, the most recent months of orders in an orders table spanning several years) to be resident in the IM column store, while using the buffer cache for the remainder of the table. With Engineered Systems such as Exadata [7], the data can therefore span multiple storage tiers: high capacity disk drives, PCI Flash cache with extremely high IOPS, and DRAM with the lowest possible response time.

Complete Transparency: The IM column store is built into the data access layer within the database engine and presents the optimizer with an alternate execution method for extremely fast table scans. The only impact to the optimizer and query processor is a different cost function for in-memory scans. By building the column store into the database engine, we ensure that all of the enterprise features of Oracle Database such as database recovery, disaster recovery, backup, replication, storage mirroring, and node clustering work transparently with the IM column store enabled.

II. New In-Memory Columnar Format

The new column format is a pure in-memory format, introduced in Oracle Database 12.1.0.2. The use of the new in-memory column format has no effect on the existing persistent format for Oracle Database. Because the column store is in-memory only, maintenance of the column store in the presence of DML changes is very efficient, requiring only lightweight in-memory data manipulation. The size of the IM column store is specified by the initialization parameter `inmemory_size`. This in-memory area is allocated from the shared memory area for a database instance (known as the *Shared Global Area* or SGA), along with other shared memory pools such as the buffer cache

A. Populating the In-Memory Column Store

The contents of the IM column store are built from the persistent contents within the row store, a mechanism referred to as *populate*. It is possible to selectively populate the IM column store with a chosen subset of the database. The new `INMEMORY` clause for `CREATE / ALTER` statements is used to indicate that the corresponding object is a candidate to be populated into the IM column store. Correspondingly, an object may be removed from the column store using the `NO INMEMORY` clause. The `INMEMORY` declaration can be performed for multiple classes of objects:

- Entire Tablespace
- Entire Table
- Entire Partition of a table
- Only a specific Sub-Partition within a Partition

An "INMEMORY advisor" is also provided that analyses historical database workloads and recommends objects to place in-memory.

Some examples of the `INMEMORY` clause are listed below in Figure 2. Note from the last example that it is also possible to exclude one or more columns in an object from being populated in-memory. In the example, the `photo` column is excluded since it is not needed by the analytical workload, and having it in memory would waste memory.

The IM column store is populated using a pool of background server processes, the size of which is controlled by the `inmemory_max_populate_servers` initialization parameter. If unspecified, the value of this parameter defaults to half the number of CPU cores.

Note that there is no application downtime while a table is being populated since it continues to be accessible via the buffer cache. In contrast, most other in-memory databases require that applications must wait till all objects are completely brought into memory before processing any user or application requests.

```
ALTER TABLE sales INMEMORY;

ALTER TABLE revenue NO INMEMORY;

CREATE TABLE customers ……
PARTITION BY LIST
  (PARTITION p1 …… INMEMORY,
  (PARTITION p2 …… NO INMEMORY);


ALTER TABLE accounts INMEMORY
NO INMEMORY (photo);
```

Figure 2: Example uses of the INMEMORY clause

Since some objects may be more important to the application than others, it is possible to control the order in which objects are populated using the `PRIORITY` subclause. By default, in-memory objects are assigned a priority of NONE, indicating that the object is only populated when it is first scanned. Note that the `PRIORITY` setting only affects the order in which objects are populated, not the speed of population. The different priority levels are enumerated in Table I below.

TABLE I
Different PRIORITY levels for in-memory populate

Priority	Description
`CRITICAL`	Object is populated immediately after the database is opened.
`HIGH`	Object is populated after all `CRITICAL` objects have been populated
`MEDIUM`	Object is populated after all `CRITICAL` and `HIGH` objects have been populated
`LOW`	Object is populated after all `CRITICAL`, `HIGH`, and `MEDIUM` objects have been populated
`NONE`	Object is only populated after it is scanned for the first time (Default)

B. In-Memory Compression

Each table selected for in-memory storage is populated into the IM column store in contiguously allocated units referred to as *In-Memory Compression Units* (IMCUs). The target size for an IMCU is a large number of rows, e.g. half a million.

Within the IMCU, each column is stored contiguously as a column *Compression Unit* (CU). The column vector itself is compressed with user selectable compression levels,

depending on the expected use case. The selection is controlled by a MEMCOMPRESS subclause.

There are basically three primary classes of compression algorithms optimized for different criteria:

1) FOR DML: This level of compression performs minimal compression in order to optimize the performance of DML intensive workloads
2) FOR QUERY: These schemes provide the fastest performance for queries while providing a 2x-10x level of compression compared to on-disk. These schemes allow data to be accessed in-place without any decompression and no additional run-time memory.
3) FOR CAPACITY: These schemes trade some performance for higher compression ratios. They require that the data be decompressed into run-time memory buffers before it can be queried. The decompression imposes a slight performance penalty but provides compression factors of 5x – 30x. The CAPACITY LOW sublevel uses an Oracle custom Zip algorithm (OZIP) that provides very fast decompression speeds; many times faster than LZO (which is the typical standard for fast decompression). OZIP is also implemented in a special-purpose coprocessor ([10], [1]) in the SPARC M7 processor.

The compression levels are enumerated in Table II below.

TABLE II
MEMCOMPRESS LEVELS FOR IN-MEMORY OBJECTS

MEMCOMPRESS	Description
DML	Minimal compression optimized for DML performance
QUERY LOW	Optimized for the fastest query performance (default)
QUERY HIGH	Also optimized for query performance, but with some more emphasis on space saving
CAPACITY LOW	Balanced between query performance and space saving (uses OZIP)
CAPACITY HIGH	Optimized for space saving

It is possible to use different compression levels for different partitions within the same table. For example in a SALES table range partitioned on time by week, the current week's partition is probably experiencing a high volume of updates and inserts, and it would be best compressed FOR DML. On the other hand, earlier partitions up to a year ago are probably used intensively for reporting purposes (including year-over-year comparisons) and can be compressed FOR QUERY. Beyond the one year horizon, even earlier partitions are probably queried much less frequently and should be compressed for CAPACITY to maximize the amount of data that can be kept in memory.

III. Query Processing

A. In-Memory Scans

Scans against the column store are optimized using vector processing (SIMD) instructions which can process multiple operands in a single CPU instruction. For instance, finding the occurrences of a value in a set of values as in Figure 3 below, adding successive values as part of an aggregation operation, etc., can all be vectorized down to one or two instructions.

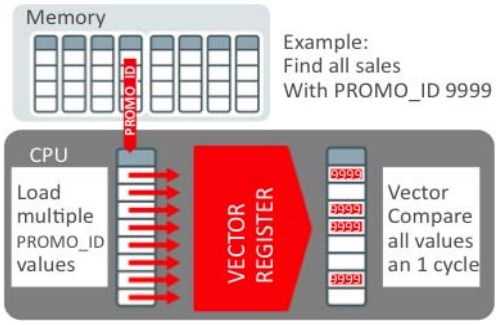

Figure 3: SIMD vector processing

Efforts are made to optimize the use of memory bandwidth by using bit-packed formats to represent the column vectors. As a result, a large number of column values can be stored within a CPU cache line.

A further reduction in the amount of data accessed is possible due to the *In-Memory Storage Indexes* that are automatically created and maintained on each of the columns in the IM column store. Storage Indexes allow data pruning to occur based on the filter predicates supplied in a SQL statement. An In-Memory Storage Index keeps track of minimum and maximum values for each column CU. When a query specifies a WHERE clause predicate, the In-Memory Storage Index on the referenced column is examined to determine if any entries with the specified column value exist in each CU by comparing the specified value(s) to the minimum and maximum values maintained in the Storage Index. If the value is outside the minimum and maximum range for a CU, the scan of that CU is avoided.

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. The metadata dictionary contains a list of the distinct values for each column within each IMCU. Thus dictionary based pruning allows Oracle Database to determine if the value being searched for actually exists within an IMCU, ensuring only the necessary IMCUs are scanned.

All of these optimizations combine to provide scan rates exceeding billions of rows per second per CPU core.

B. In-Memory Joins

Apart from accelerating scans, the IM column store also provides substantial performance benefits for joins. Typical

star joins between fact and dimension tables can be reduced to a filtered scan on a dimension table to build a compact Bloom filter of qualifying dimension keys (Figure 4), and then a subsequent scan on the fact table using the Bloom filter to eliminate values that will not be join candidates. The Bloom filter thus greatly reduces the number of values processed by the join. While this method is not specific to in-memory, the optimizer is now able to select this method more frequently due to the massive reduction in the underlying table scan costs.

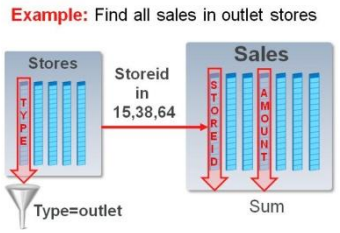

Figure 4: Bloom filter

C. *In-Memory Aggregations and Groupings*

The basic primitives of very fast scans and joins can be further leveraged to accelerate complex reports featuring multiple levels of aggregations and groupings. A new optimizer transformation, called *Vector Group By*, is used to compute multi-dimensional aggregates in real-time. The Vector Group By transformation is a two-part process similar to the well known star transformation.

Let's consider the following query as an example, which lists the total revenue from sales of footwear products in outlet stores, grouped by store and by product:

```
Select Stores.id, Products.id, sum(Sales.revenue)
From Sales, Stores, Products
Where Sales.store_id = Stores.id
And Sales.product_id = Products.id
And Stores.type = "Outlet"
And Products.type = "Footwear"
Group by Stores.id, Products.id;
```

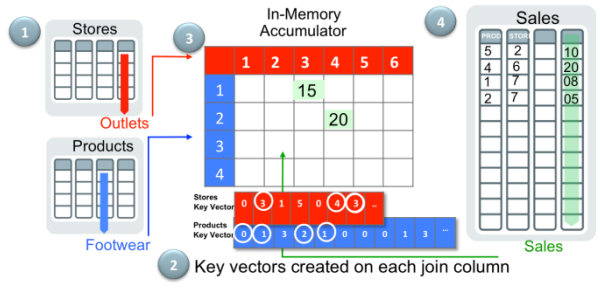

Figure 5. Vector Group By example

The execution steps (depicted in Figure 5) are as follows:

Step 1: The Query begins by scanning the STORES and PRODUCTS dimensions.

Step 2: A new data structure called a *Key Vector* is created based on the results of each of these scans. A key vector maps a qualifying dimension key to a dense grouping key (e.g. each store that is of type "outlet" will have a non-zero grouping key numbered 1..N, while each product that is of type "footwear" will have a non-zero grouping key numbered 1..M)

Step 3: The key vectors are then used to create an additional multi-dimensional array known as an *In-Memory Accumulator*. Each dimension of the In-Memory accumulator will have as many entries as the number of non-zero grouping keys corresponding to that dimension; in the above example, it will be an (NxM) array.

Step 4: The second part of the execution plan is the scan of the SALES table and the application of the key vectors. For each entry in the SALES table that matches the join conditions based on the key vector entries for the dimension values (i.e. has a non-zero grouping key for each dimension), the corresponding sales amount will be added to the appropriate cell in the In-Memory Accumulator. At the end, the In-Memory Accumulator will have the results of this aggregation.

The combination of these two phases dramatically improves the efficiency of a multiple table join with complex aggregations. Again, this is not an in-memory specific optimization, but one that can be frequently chosen by the optimizer in view of the massive speedup of scans.

IV. Transactional Consistency

The default isolation level provided by Oracle Database is known as *Consistent Read*, described in [6]. With Consistent Read, every transaction in the database is associated with a monotonically increasing timestamp referred to as a *System Change Number* (SCN). A multi-versioning protocol is employed by the buffer cache to ensure that a given transaction or query only sees changes made by transactions with older SCNs.

The IM column store similarly maintains the same consistent read semantics as the buffer cache. Each IMCU is marked with the SCN of the time of its creation. An associated metadata area, known as a Snapshot Metadata Unit (SMU) tracks changes to the rows within the IMCU made beyond that SCN.

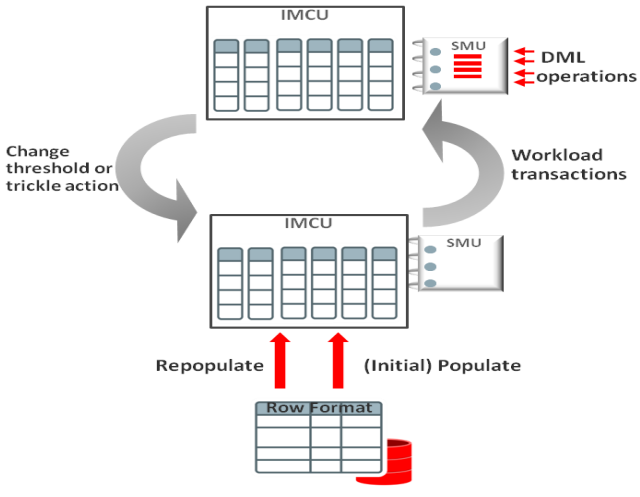

Figure 6: Change tracking and repopulate

The SMU tracks the validity of rows within the IMC. Changes made by transactions are processed as usual via the Buffer Cache, but for tables populated into the IM column

store, the changes made to them are also logged in an in-memory "transaction journal" within the SMU.

When a column scan runs against the base IMCU, it fetches the changed column values from the journal for all the rows that have been modified within the IMCU at an SCN earlier than the SCN of the scan (according to Consistent Read semantics, changes made at an SCN higher than the SCN of the scan are not visible to the scan).

As changes accumulate in the journal, retrieving data from the transaction journal causes increased overhead for scans. Therefore, a background *repopulate* mechanism periodically rebuilds the IMCU at a new SCN, a process depicted in Figure 6 above. There are two basic kinds of repopulate:

1. **Threshold repopulate**: Threshold repopulate is a heuristic driven mechanism that repopulates an IMCU once the changes to the IMCU exceed a certain threshold and various other criteria are met.
2. **Trickle Repopulate**: Trickle repopulate runs constantly and unobtrusively in the background, consuming a small fraction of the available repopulate server processes. The goal of trickle repopulate is to ensure that eventually any given IMCU will be completely clean even if it is had not been sufficiently modified to qualify for threshold repopulate. The number of populate background processes that can be used for trickle repopulate is controlled by another initialization parameter: `inmemory_trickle_repopulate_servers_percent`. This parameter defaults to 1. For example, if `inmemory_max_populate_servers = 10`, at most one-tenth of a single CPU core is dedicated to trickle repopulate.

V. In-Memory Column Store Scale-Out

Apart from providing in-memory columnar storage on a single machine, Oracle Database In-Memory can scale out across multiple machines using the Oracle Real Application Cluster (RAC) configuration, featuring a protocol known as *Cache Fusion* [8].

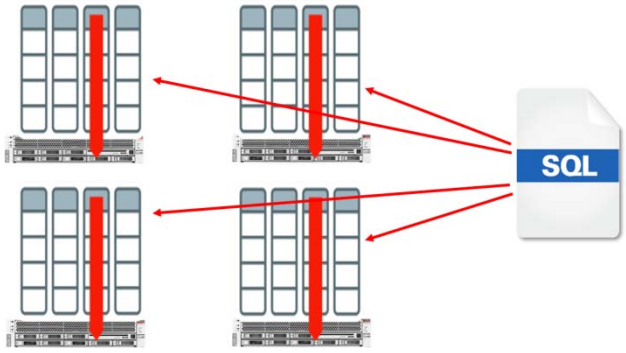

Figure 7: In-Memory Scale-Out

On a RAC cluster, queries on in-memory tables are automatically parallelized across multiple instances as depicted in Figure 7 above. This results in each parallel query process accessing local in-memory columnar data on each instance, with each process then only having to perform a fraction of the work.

A. Distribution and Parallel Execution

The user can specify how the data within a table is to be distributed across the instances of a RAC cluster by using the `DISTRIBUTE` clause. There are four possible options for thus distributing the contents of a table:

1. `DISTRIBUTE BY PARTITION`: When a table is partitioned, this distribution scheme assigns all data from the same partition to the same instance and maps different partitions to different instances. This type of distribution is specially suited to hash partitions which usually exhibit uniform access across partitions. Distributing by partition has another potential benefit: It can allow in-memory *partition-wise joins* between two tables that are partitioned on the same attribute. For instance, if a `SALES` table is hash partitioned on `order_id`, and a `SHIPMENTS` table is likewise hash partitioned by `order_id`, the `SALES` to `SHIPMENTS` join can be accomplished as a set of local in-memory joins on each instance since the partitions will be co-located.
2. `DISTRIBUTE BY SUBPARTITION`: In Oracle Database, tables can be composite partitioned: i.e. Partitioned by some criteria, and then each partition further sub-partitioned by a different criteria. Distributing by sub-partition can be helpful when the top-level partitioning criteria could cause skewed data access. For instance if a `SALES` table is partitioned by weekly date ranges, it is likely that distributing by partition would cause skewed access, since more recent date ranges would probably exhibit far more activity. However if the table were further sub-partitioned by hash on `order_id`, distributing by sub-partition would provide uniform accesses across instances. Furthermore the above join between `SALES` and `SHIPMENTS` could still be done as in-memory local joins, since all the `SALES` sub-partitions will be collocated with all the `SHIPMENTS` partitions with the same `order_id` hash.
3. `DISTRIBUTE BY ROWID RANGE`: When a table is not partitioned, or the partitioning criteria used would cause severely skewed access, then it is possible to ensure that the table contents are uniformly distributed across the instances using `ROWID RANGE` based distribution – which places an IMCU on an instance by applying a uniform hash function to the rowid of the first row within that IMCU. This scheme therefore distributes the contents of the table without regard to the actual values. While this does result in uniform distribution of the table contents and uniform accesses across the instances, it does have the drawback of precluding partition-wise joins.
4. `DISTRIBUTE AUTO`: 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 partitioning criteria and optimizer statistics.

B. In-Memory Fault Tolerance

On Oracle Engineered systems such as Exadata and SPARC Supercluster, a fault-tolerance protocol specially optimized for high-speed networking (direct-to-wire Infiniband) is used to allow in-memory fault tolerance as depicted in Figure 8 below.

Fault-tolerance is specified by the `DUPLICATE` subclause. For a table, partition, or sub-partition marked as `DUPLICATE`, all IMCUs for that object are populated in two instances. Having a second copy ensures that there is no downtime if an instance fails since the second copy is instantly available. Both IMCUs are master copies that are populated and maintained independently, and both can be used at all times for query processing.

Figure 8: In-memory column store fault tolerance

For small tables containing just a few IMCUs (such as small dimension tables that frequently participate in joins), it is advantageous to populate them on every instance, in order to ensure that all queries obtain local access to them at all times. This full duplication across all instances is achieved using the `DUPLICATE ALL` subclause.

VI. Conclusions and Future Work

While column oriented storage is known to provide substantial speedup for Analytics workloads, it is also unsuited to OLTP workloads characterized by high update volumes and selective queries. The Oracle Database In-Memory Option provides a true dual-format in-memory approach that is seamlessly built into the Oracle Data Access Layer, which allows it to be instantly compatible with all of the rich functionality of the Oracle Database. In-Memory storage can be selectively applied to important tables and to recent partitions, while leveraging the mature buffer cache, flash storage and disk for increasingly colder data.

As a result of this balanced approach, Oracle Database In-Memory allows the use of in-memory technology without any of the compromises or tradeoffs inherent in many existing in-memory databases.

Future work includes integrating Oracle Database In-Memory into the Automatic Data Optimization (ADO) framework (introduced with Oracle Database 12c) that automatically migrates objects between different storage tiers based on access frequency, the ability to create the in-memory column store on an Active Dataguard (physical standby) Database (in the 12.1.0.2 release it is supported on Logical Standby databases but not on a Physical Standby), and extending the in-memory columnar format to Flash and other emerging persistent memory technologies.

Acknowledgements

The authors represent a subset of the large team that delivered Oracle Database In-Memory, and they acknowledge the contributions of many others in the Oracle Database organization in the areas of Data, Space and Transaction Management, Optimizer, Parallel Execution, RAC, OLAP, Virtual OS, as well as the Performance and QA teams.

The authors also acknowledge the Oracle SPARC hardware organization for their many contributions in optimizing their processor designs for Database In-Memory workloads and for implementing custom silicon functionality specifically for Database In-Memory operations such as vectorized column scans and OZIP decompression.

References

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

[2] T. Lahiri, M. Neimat, and S. Folkman, "Oracle TimesTen: An In-Memory Database for Enterprise Applications", *IEEE Data Engineering Bulletin 36, no. 2 (2013)*, pp. 6-13, 2013.

[3] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones et al. "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System", *Proceedings of the VLDB Endowment 1, no. 2,* pp. 1496-1499, 2008.

[4] 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.

[5] M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau et al, "C-Store: A Column-Oriented DBMS", in *Proceedings of VLDB ‘05*, pp. 553-564., 2005.

[6] 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.

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

[8] T. Lahiri, V. Srihari, W. Chan, N. Macnaughton, and S. Chandrasekaran, "Cache Fusion: Extending Shared-Disk Clusters with Shared Caches", in *Proceedings of VLDB ‘01*, pp. 683-686, 2001.

[9] R. Ramamurthy, D. DeWitt, and Q. Su, "A Case for Fractured Mirrors", *The VLDB Journal - the International Journal on Very Large Data Bases 12 no.2,* pp. 89-101, 2003.

[10] S. Phillips, "M7: Next Generation SPARC", in *Hot Chips 26*, Cupertino, 2014.

[11] K. Aingaran, S. Jarath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, T. Wicki, "M7: Oracle’s Next Generation SPARC Processor", (to appear) in *Micro, IEEE, vol. 35, no. 2*, 2015.