In-Memory Database Systems (IMDSs): Pushing Past the Terabyte-Plus Boundary

A BENCHMARK REPORT

Abstract: In-memory database systems (IMDSs) hold out the promise of breakthrough performance for time-sensitive, data-intensive tasks. Yet IMDSs' compatibility with very large databases (VLDBs) has been largely uncharted. This benchmark analysis fills the information gap and pushes the boundaries of IMDS size and performance. Using McObject's 64-bit *eXtreme*DB® technology, the application creates a 1.17 Terabyte, 15.54 billion row database on a 160-core Linux-based SGI® Altix® 4700 server. It measures time required for database provisioning, backup and restore. In SELECT, JOIN and SUBQUERY tests, benchmark results range as high as 87.78 million query transactions per second. The report also examines efficiency in utilizing all of the test bed system's 160 processors and includes full database schema and relevant application source code.

McObject LLC
33309 1st Way South
Suite A-208
Federal Way, WA 98003

Phone: 425-888-8505
E-mail: info@mcobject.com
www.mcobject.com

Introduction

The Louisiana Immersive Technologies Enterprise (LITE), hosted in the Research Park of the University of Louisiana at Lafayette, is a facility designed for exploring technological boundaries. LITE recently added a massive computing component to its already world-leading infrastructure: a 160-processor SGI® Altix® 4700 server with 4.1 terabytes of memory.

The acquisition positions LITE in the top echelon of computing resources available for private and public sector projects. The facility targets the most demanding analytical tasks: seismic modeling for energy exploration, signals and imagery intelligence for the military, business intelligence, real-time automobile impact simulations, geospatial analysis, and many other projects that require crunching vast amounts of data in limited time.

In short, LITE is ideally equipped as a proving ground for the kinds of performance-intensive technology sought by forward-looking companies to gain competitive advantage.

One such technology is the in-memory database system (IMDS), a type of database management system (DBMS) software used in high performance applications including data analytics, securities trading, telecommunications, real-time military/aerospace, embedded systems, and science and engineering applications. IMDSs eliminate disk access, storing data in main memory and sending changes to the system's hard disk (if there is one) only when specified by the application. In contrast, traditional 'on-disk' DBMSs cache frequently requested data in memory for faster access, but automatically write data updates, insertions, and deletes through to disk—a requirement that imposes often unacceptable levels of mechanical and logical overhead on application performance.

IMDSs' streamlined, all-in-memory operation makes them very fast. Because IMDSs are relatively new (commercial versions emerged some 10 years ago), the technology's boundaries are still being explored. Is there a maximum size, measured in bytes or database rows/records, for an in-memory data store? What are the outer limits of IMDS performance? The goal of this benchmark test is to explore these IMDS boundaries.

LITE's sheer computing horsepower qualifies the facility for IMDS benchmarking, and its choice of SGI Altix servers, with their efficient memory use and sophisticated multi-processing support, make it an especially valuable test bed. Altix offers the industry's most scalable system for global shared memory—up to 24TB of globally addressable memory in a system. In addition, the SGI NUMAlink Interconnect Fabric allows multiple CPU nodes to be tightly integrated, so that all the pieces of memory are seen as one, and in-memory database queries and other operations can be spread evenly across multiple processors.

Benchmark Goals

The goals for the benchmark were to demonstrate that a 64-bit in-memory database system could provision a terabyte-size database efficiently (no IMDS vendor has published a benchmark involving a terabyte or more of data), and that every available CPU core could be exercised in parallel to run multiple queries and maintain nearly linear performance scalability (i.e. if one processor core can execute a given query in 2 microseconds, then the same query could be executed on 160 processor cores simultaneously with elapsed time as close to 2 microseconds as possible).

The test environment was LITE's SGI Altix 4700 system with 80 dual-core 1.6 Ghz Itanium 2 processors (160 cores total) and 4 terabytes of RAM. LITE provided access to the system, and the IMDS used for the benchmark was the 64-bit version of McObject's *eXtreme*DB® in-memory database system. McObject launched *eXtreme*DB in 2001 and its customers use the IMDS in software applications ranging from securities trading systems to military/aerospace equipment and industrial control. The 64-bit operating system for the benchmark test was SUSE Linux Enterprise Server 9, optimized with SGI's ProPack software for performance and stability.

Benchmark Application

For the benchmark, engineers created a simple database structure consisting of two tables: a PERSONS table and an ORDER table. These tables represent the two sides of any generic transaction such as the examples given in Listing 1 below. In all such cases, there are two instances of a 'person' (one for each side of the transaction) and one instance of an 'order' that represents the transaction between the two entities.

ORDER table role	1st PERSON instance role	2nd PERSON instance role
Gift	giver	receiver
Trade Execution	buyer	seller
Call Record	caller	callee

Listing 1 – the roles of ORDER and PERSON tables in different types of transactions.

Obviously, this database structure can be embellished to include any sort of demographic or other information that would help to increase the application's business value.

Following are examples of the PERSONS and ORDERS tables.

PERSON ID	NAME	ADDRESS
1	Steve Graves	Issaquah, WA
2	Andrei Gorine	Issaquah, WA
3	Ted Kenney	Issaquah, WA

Listing 2 – example data of the PERSONS table.

ORDER ID	SENDER ID	RECEIVER ID	TOTAL ORDERS	FIRST ORDER DATE	LAST ORDER DATE
1	1	2	2	11/1/2006	11/30/2006
2	1	3	2	11/1/2006	11/30/2006
3	2	1	1	11/1/2006	11/1/2006

Listing 3 – example data of the ORDERS table

The simple data definition in native *eXtreme*DB DDL syntax is shown in figure 1, and in SQL syntax in figure 2.

```
#define uint8     unsigned<8>
declare database spdb;

class persons {
      uint8       PERSON_ID;
      char<30>    NAME;
      char<30>    ADDRESS;
      hash<PERSON_ID> hkey[6000000000];
};

class orders {
      uint8 ORDER_ID;
      uint8 SENDER_ID;
      uint8 RECEIVER_ID;
      uint8 TOTAL_ORDERS;
      date FIRST_ORDER_DATE;
      date LAST_ORDER_DATE;
      hash<ORDER_ID> hkey[25000000000];
      voluntary tree<SENDER_ID,RECEIVER_ID> hkey2;
      voluntary tree<SENDER_ID>     hkey3;
};
```

Figure 1.

```
CREATE TABLE persons(
        PERSON_ID BIGINT PRIMARY KEY UNIQUE,
        NAME char(30),
        ADDRESS char(30)
)

CREATE TABLE orders(

        ORDER_ID BIGINT PRIMARY KEY UNIQUE,
        SENDER_ID BIGINT,
        RECEIVER_ID BIGINT,
        TOTAL_ORDERS BIGINT,
        FIRST_ORDER_DATE datetime,
        LAST_ORDER_DATE datetime
)
CREATE INDEX hkey2 ON orders ( SENDER_ID )

CREATE INDEX hkey3 ON orders ( SENDER_ID,RECEIVER_ID )
```

Figure 2.

Regardless of whether the database is defined with the native *eXtreme*DB DDL or SQL DDL, the database can be accessed by either *eXtreme*DB's native application programming interface or by its SQL API (*eXtreme*SQL). The choice to define the database schema in one DDL or the other is based on the database designer's preference and will not affect database application performance.

To populate the database, engineers created an array of 30,000 random strings and selected random elements from the array to populate the NAME and ADDRESS columns. In the benchmark application, unique values for PERSON_ID and ORDER_ID are generated sequentially, optionally from a starting value supplied on the command line.

The number of PERSON objects created is specified on the command line. For this test, engineers created 3 billion PERSONS records (rows) and 12.54 billion ORDERS records (rows), resulting in a database size of 1.17 terabytes.

Does this constitute a very large database? Consider that the Winter Corporation, a research and consulting firm, surveys major corporations including Amazon.com, Dell and AT&T annually to identify the world's largest and most heavily used databases. In Winter Corporation's 2005 survey results, as published on its Web site[i], a 1.229 terabyte database ranks as the fourth largest Linux on-line transaction processing (OLTP) database. A 12.54 billion row database, such as the one created for this benchmark, would rank as the second-largest in this category (Linux OLTP) of Winter Corporation's survey.

The inescapable conclusion is that a database consisting of a 3 billion row table and a 12.54 billion row table is a very large database.

All of the benchmark implementation code can be executed using the native *eXtreme*DB programming interface or McObject's implementation of the SQL database language.

The native interface executes substantially faster than SQL by avoiding the overhead of SQL parsing, optimization and execution. The native interface goes directly to the table and navigates the database indexes and data via an intuitive, type-safe programming interface that is generated when the DDL file is processed by the schema compiler. For straightforward queries, the native interface can be as easy, and in some cases easier, to program as the SQL API.

Though slower due to the necessary parsing, optimization and execution steps, the SQL implementation has the advantage of familiarity – SQL is known by many programmers – and of being potentially easier to program, especially for complex queries involving many joins, aggregation, and/or complex filters, or when the result set must be sorted in an order that is not directly supported by indexes defined in the DDL. (The latter advantage of SQL would only occur in situations requiring ad hoc queries. When queries are known in advance, database designers can create appropriate indexes.)

*eXtreme*DB offers programmers the choice: the unsurpassed performance of a direct native interface, or the convenience of SQL for complex queries or when flat-out performance is not required.

Performance – Ingest

The following graphs depict the performance of provisioning the *eXtreme*DB database with the 15.54 billion rows of test data.

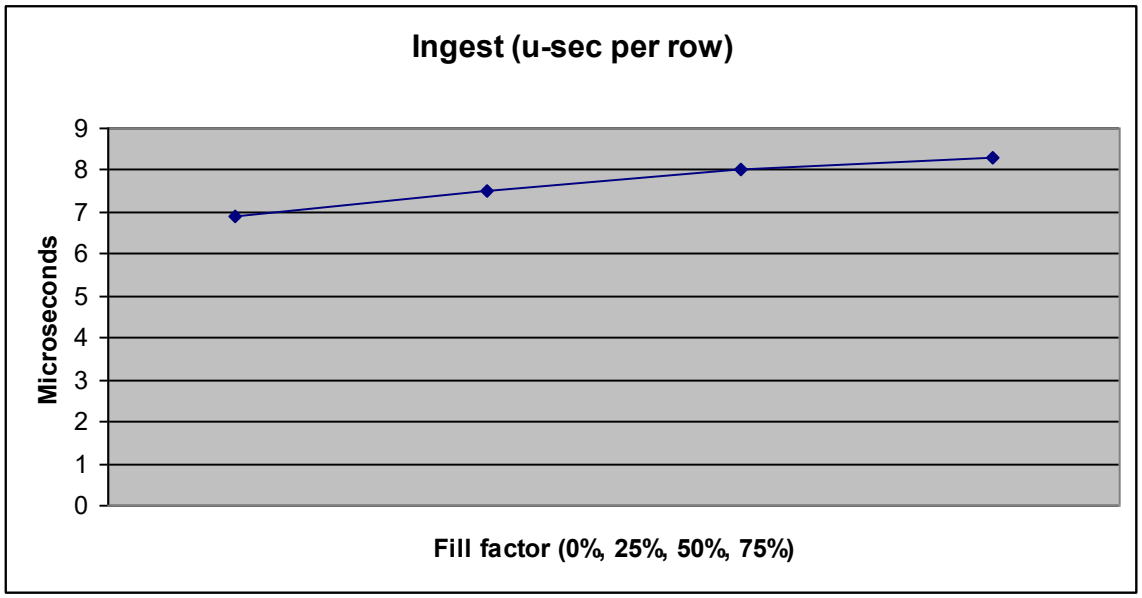

Chart 1 – Microseconds per row ingest performance.

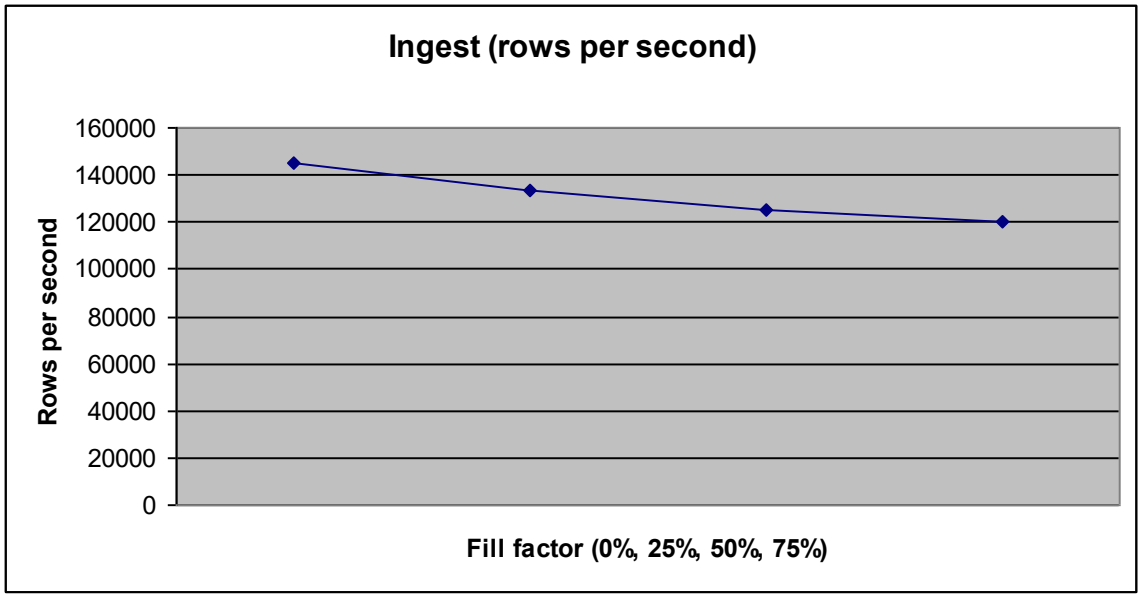

Chart 2 – Rows per second ingest performance.

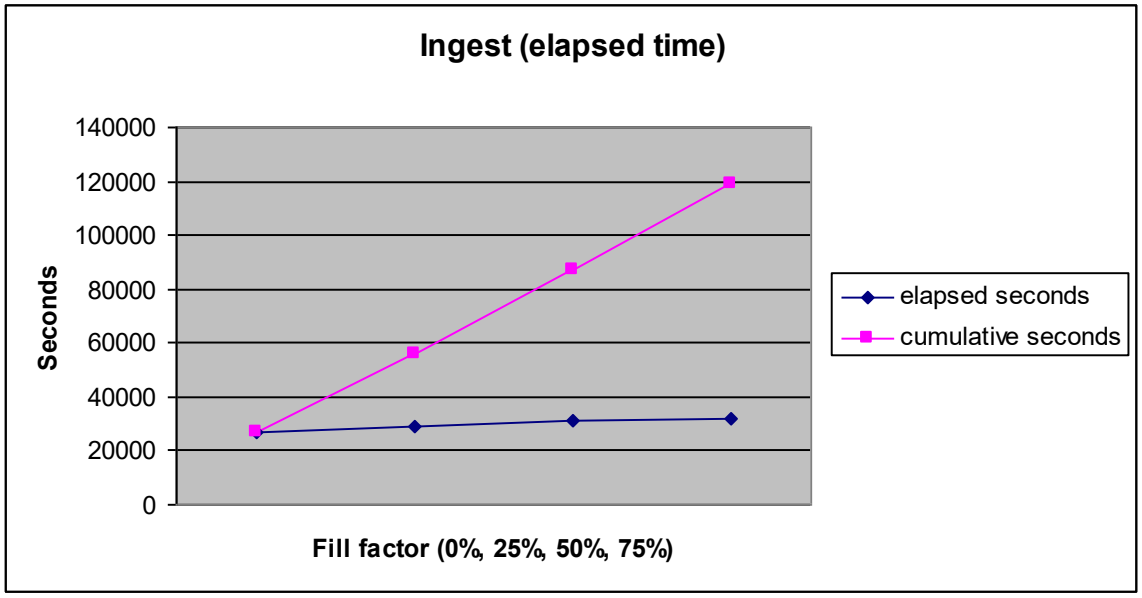

Chart 3 – Ingest performance: Elapsed time.

The total time to provision the 1.17 terabyte, 15.54 billion row database was just over 33 hours. The per-row insert time for the last quartile of data was a very respectable 8.3 microseconds. The per-row insert time for the first quartile of data was 6.9 microseconds. Ingest performance between first and last quartile decreased by just 20 percent – much less than the precipitous performance drop-off that is often predicted for the later stages of populating a very large database.[ii]

It is important not to confuse the time required to ingest a data set, with the time needed to back up the data once it has been loaded, or to restore it after potential failure. As part of this benchmark test, engineers backed up the fully provisioned in-memory database in

4.3 hours, and restored it in 4.76 hours. So, while initial ingest took 33 hours, the database could be saved and reloaded for subsequent use in a fraction of that time. (And, once reloaded, it can be extended with new data.) Backing up and restoring the provisioned database is a simple matter of streaming the in-memory image to persistent storage; there is no need to allocate pages, assign records to pages, maintain indexes, etc., so back up and restore performance is largely a function of the speed of the persistent media.

Backup and restore capabilities are especially important when working with in-memory database systems because these functions are the primary means to achieve data persistence. While IMDSs have persistence mechanisms such as transaction logging, the types of applications served by IMDSs for data analysis have data persistence needs that differ from most mainstream enterprise systems. Data mining, modeling and other analytics applications exist to process data in its transient state, rather than provide long-term storage. Therefore backup and restore functions are generally considered sufficient to provide persistence for such applications.

Performance – Select, Join and Subquery

The following tables depict the performance of *e*X*treme*DB executing queries against the 1.17 terabyte database consisting of 3 billion rows of the PERSONS table and 12.54 billion rows of the ORDERS table. On average, each row of the PERSONS table has 4.18 related ORDERS rows (min = 1, max = 22).

The tests were run with varying numbers of threads (1, 80 and 160).

The first test was a simple query:

```
SELECT name FROM persons WHERE person_id = ?
```

The code in the *e*X*treme*DB native interface is shown in figure 3, and the SQL interface in figure 4.

```
int rawSelect(mco_db_h pdb, mco_trans_h t,uint8 pID)
{
      mco_trans_h pt=t;
      persons P;
      char buf[50];
      uint4 rCnt=0;
      int rs=MCO_S_OK;
      mco_puint l0;

      if (!pt)
            if ((rs = mco_trans_start( pdb, MCO_READ_ONLY,
                                 MCO_TRANS_FOREGROUND, &pt))!= MCO_S_OK)
            {
                  printf("Cant create transaction!\n");
                  return rs;
            }
      if (persons_hkey_find(pt, pID, &P) == MCO_S_OK)
      {
            // load person_name
            persons_NAME_get(&P,buf,50);
            if (rs == MCO_S_OK) rCnt++;
      }
      if (!t)
            mco_trans_commit(pt);
      return rCnt;
}
```

Figure 3 – *eXtreme*DB native programming interface implementation of a simple SELECT.

```
void doSelect(int tid, void* pThrParam, mco_trans_h t, uint4 cnt)
{
    uint4 i;
    STAT_DATA_R* pstat=0;

    SQLHSTMT hStmt;
    SQLHDBC hDbc = (SQLHDBC)pThrParam;
    uint8 personId;
    SQLAllocStmt(hDbc, &hStmt); // allocate statement
    SQLPrepare(hStmt,
              (SQLCHAR*)"SELECT NAME FROM persons WHERE PERSON_ID = ?",
               SQL_NTS); // prepare query
    SQLBindParameter(hStmt, 1, SQL_PARAM_INPUT, SQL_C_UBIGINT,
                     SQL_BIGINT, 0, 0, &personId, 0, NULL);

    for (i = 0;i < cnt; i++)
    {
        personId = getPerson(tid);
        SQLExecute(hStmt); // execute prepared statement
        SQLCloseCursor(hStmt); // release result set

    }
    SQLFreeStmt(hStmt, SQL_DROP); // drop statement
}
```

Figure 4 – *eXtreme*DB SQL implementation of a simple SELECT.

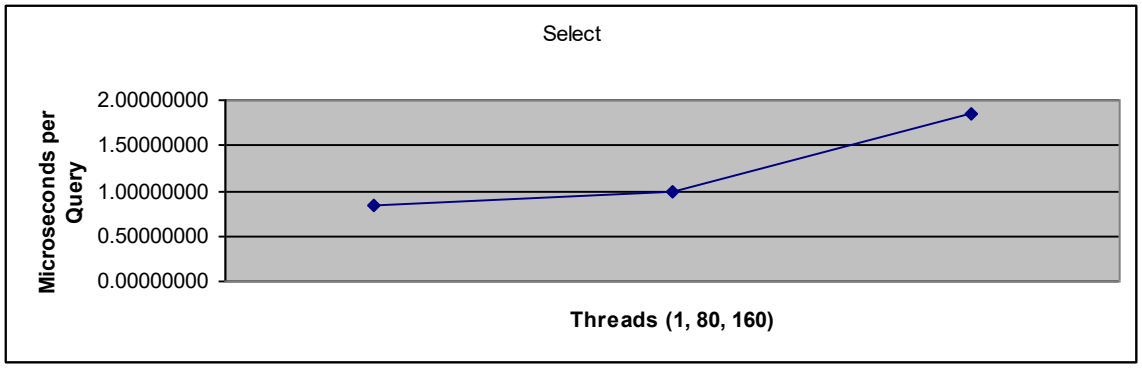

Chart 4 – SELECT performance of the *eXtreme*DB native interface in microseconds-per-query.

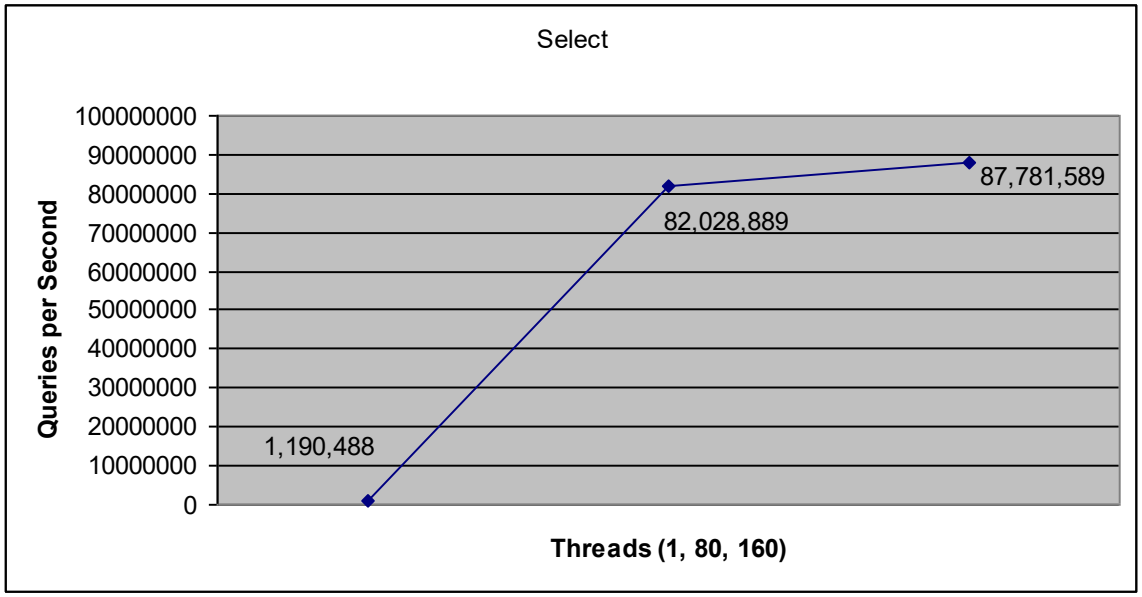

Chart 5 – SELECT performance of the *eXtreme*DB native interface in queries-per-second.

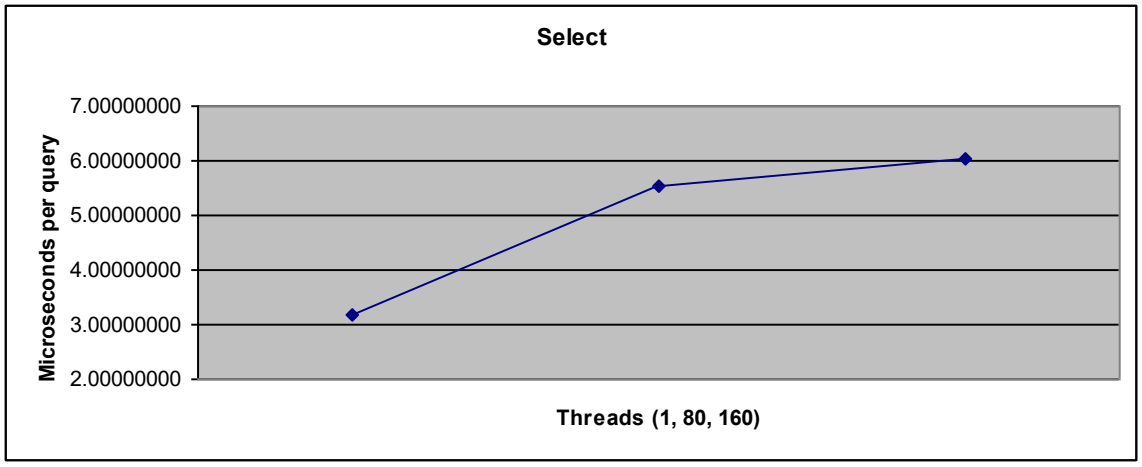

Chart 6 – SELECT performance of SQL ODBC API in microseconds-per-query.

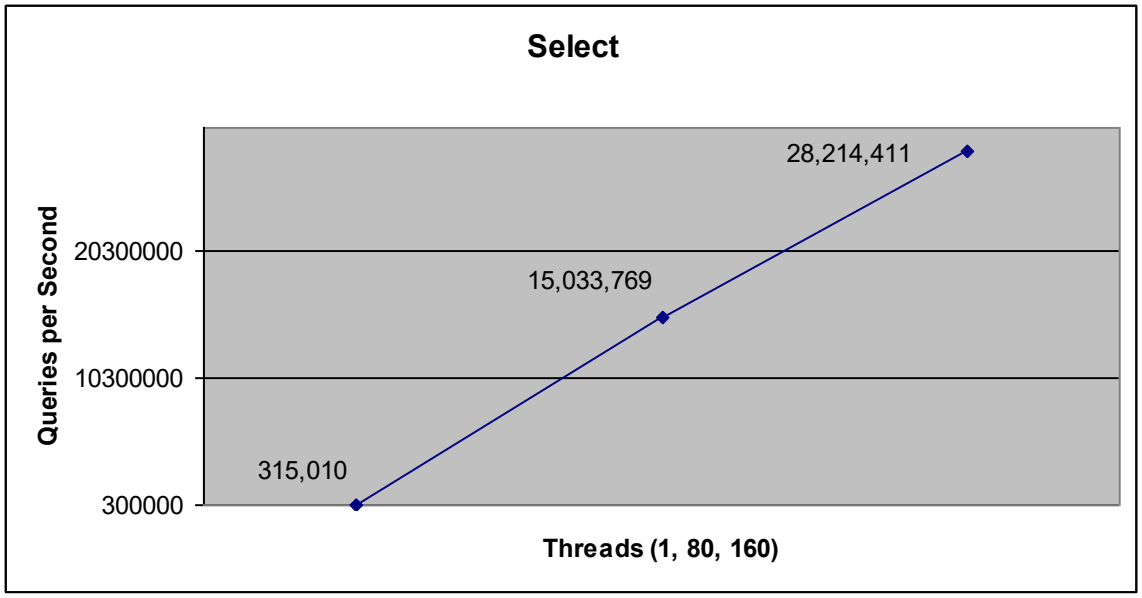

Chart 7 – SELECT performance of *eXtreme*DB SQL in queries-per-second.

Just how fast is a performance of 87.7 million query transactions per second, delivered by the *eXtreme*DB 64-bit in-memory database system for Linux, using its native API and running on SGI's 160-core server? And how fast is the 28.2 million queries per second achieved using *eXtreme*DB's ODBC SQL API?

Comparing transaction processing performance between different applications in different operating environments is notoriously tricky. But to put the benchmark result of 87.7 million queries per second in perspective, consider that the "standard currency" of such comparisons is *transactions per minute*. For example, Microsoft in late 2005 announced the SQL Server database's new "world record" in Microsoft Windows-based on-line transaction processing of more than 1 million transactions per minute (or 16,666.67 transactions per second). "SQL Server has once again demonstrated that it is capable of handling the most demanding online transaction processing (OLTP) workloads," the company stated on its Web site, announcing SQL Server's entry into the "1 million transactions per minute club." We provide this comparison for illustration purposes only and it is not intended to suggest that the benchmark described herein is similar to the TPC-C or TPC-H benchmarks referenced in Microsoft's announcement.

The second test was a query with a three-table join:

SELECT A.NAME, C.NAME, B.ORDER_ID, B.TOTAL_ORDERS, B.FIRST_ORDER_DATE, B.LAST_ORDER_DATE FROM persons A, orders B, persons C WHERE A.PERSON_ID = ? AND A.PERSON_ID = B.SENDER_ID AND C.PERSON_ID = B.RECEIVER_ID

The code in the *eXtreme*DB native interface is shown in figure 4, and the *eXtreme*DB SQL interface in figure 5.

```
int rawJoin(mco_db_h pdb,mco_trans_h t,uint8 pID,
            struct STAT_DATA_R** pstat)
{
      mco_trans_h pt=t;
      orders T; //B
      persons P1; //A
      persons P2; //C
      uint8 rcv;
      mco_cursor_t cr;
      int rss;
      uint4 rCnt=0;
      int rs=MCO_S_OK;

      if (!pt)
            if (( rs = mco_trans_start(pdb, MCO_READ_ONLY,
                        MCO_TRANS_FOREGROUND, &pt)) != MCO_S_OK)
            {
                  printf("Can't create transaction!\n");
                  return rs;
            }
      if (persons_hkey_find(pt, pID, &P1) == MCO_S_OK)
      {
            // person with ID found ...
            // searching for sender_id==% records...

            if (orders_hkey3_index_cursor(pt, &cr) == MCO_S_OK)
            {
                  rs = orders_hkey3_search(pt,&cr,MCO_EQ,pID);
                  while(rs == MCO_S_OK)
                  {
                        // load receiver_id && hash search
                        orders_from_cursor(pt, &cr, &T);
                        rs = orders_hkey3_compare(pt, &cr, pID, &rss);
                        if(rs == MCO_S_OK && rss != 0)
                              break;

                        orders_RECEIVER_ID_get(&T, &rcv);
                        if(rs == MCO_S_OK &&
                           persons_hkey_find(pt, rcv, &P2) == MCO_S_OK)
                              rCnt++;
                        if(rs == MCO_S_OK)
                              rs = mco_cursor_next(pt, &cr);
                  }
            }
      }
      if (!t)
             mco_trans_commit(pt);

      return rCnt;
}
```

Figure 5 – *eXtreme*DB native programming interface implementation of a three-table JOIN.

```
void doJoin(int tid, void* pThrParam, mco_trans_h t, uint4 cnt)
```

```
{
    uint4 i;
    STAT_DATA_R* pstat=0;

    SQLHDBC hDbc = (SQLHDBC)pThrParam;
    SQLHSTMT hStmt;
    uint8 personId;
    SQLAllocStmt(hDbc, &hStmt); // allocate statement
    SQLPrepare(hStmt,
               (SQLCHAR*)"SELECT A.NAME, C.NAME, B.ORDER_ID,"\
" B.TOTAL_ORDERS, B.FIRST_ORDER_DATE, B.LAST_ORDER_DATE FROM"\
" persons A, orders B, persons C WHERE A.PERSON_ID = ? AND "\
" A.PERSON_ID = B.SENDER_ID AND C.PERSON_ID = B.RECEIVER_ID",
                SQL_NTS); // prepare query
    SQLBindParameter(hStmt, 1, SQL_PARAM_INPUT, SQL_C_UBIGINT,
                     SQL_BIGINT, 0, 0, &personId, 0, NULL);

    for (i=0;i<cnt;i++)
    {
        personId = getPerson(tid);
        SQLExecute(hStmt); // execute prepared statement
        SQLCloseCursor(hStmt); // release result set
    }
    SQLFreeStmt(hStmt, SQL_DROP); // drop statement
}
```

Figure 6 – *eXtreme*DB ODBC implementation of a three-table JOIN.

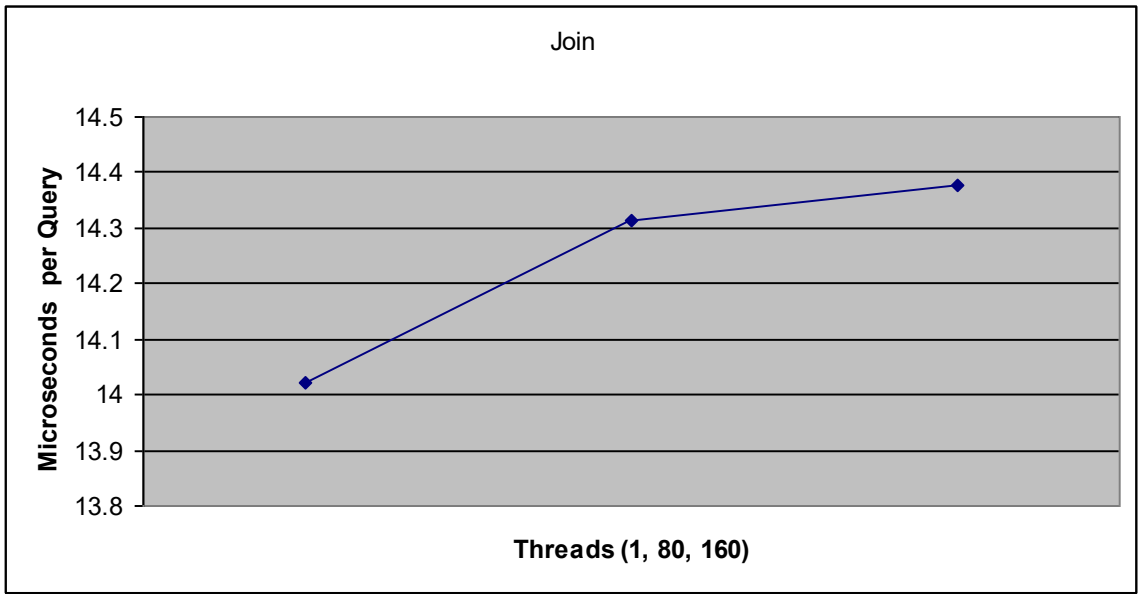

Chart 8 – JOIN performance of the *eXtreme*DB native interface in microseconds-per-query.

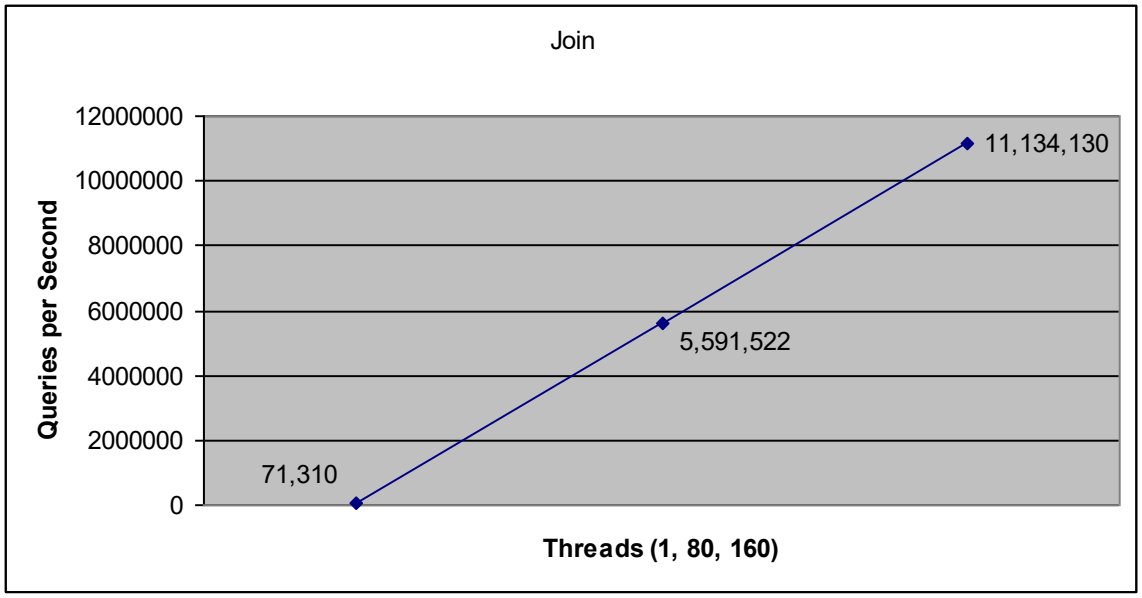

Chart 9 – JOIN performance of the *eXtreme*DB native interface in queries-per-second.

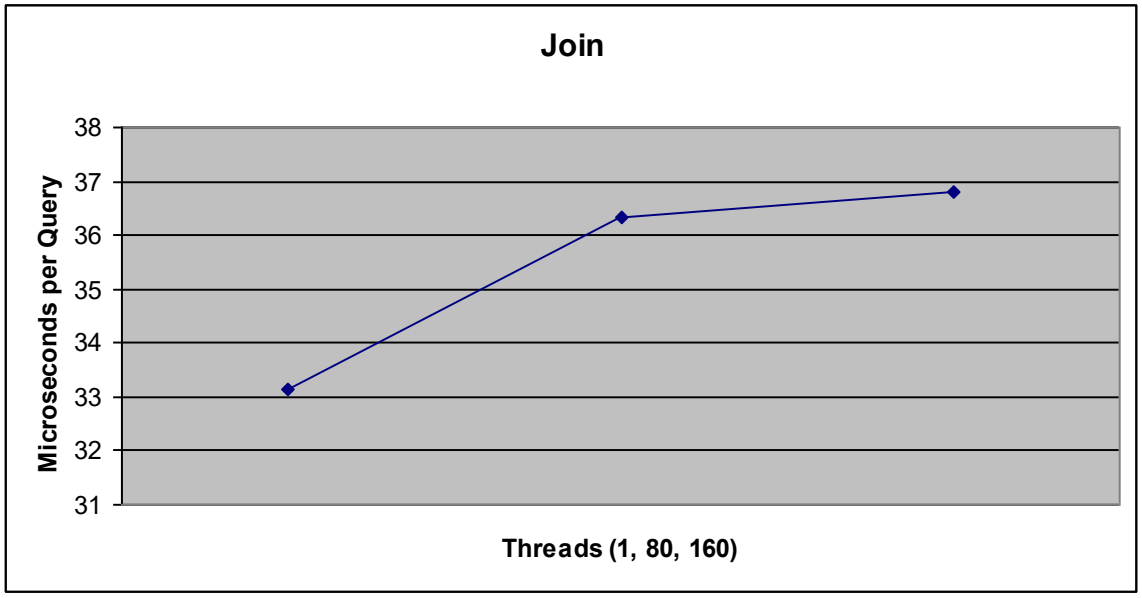

Chart 10 – JOIN performance of the *eXtreme*DB SQL ODBC interface in microseconds-per-query.

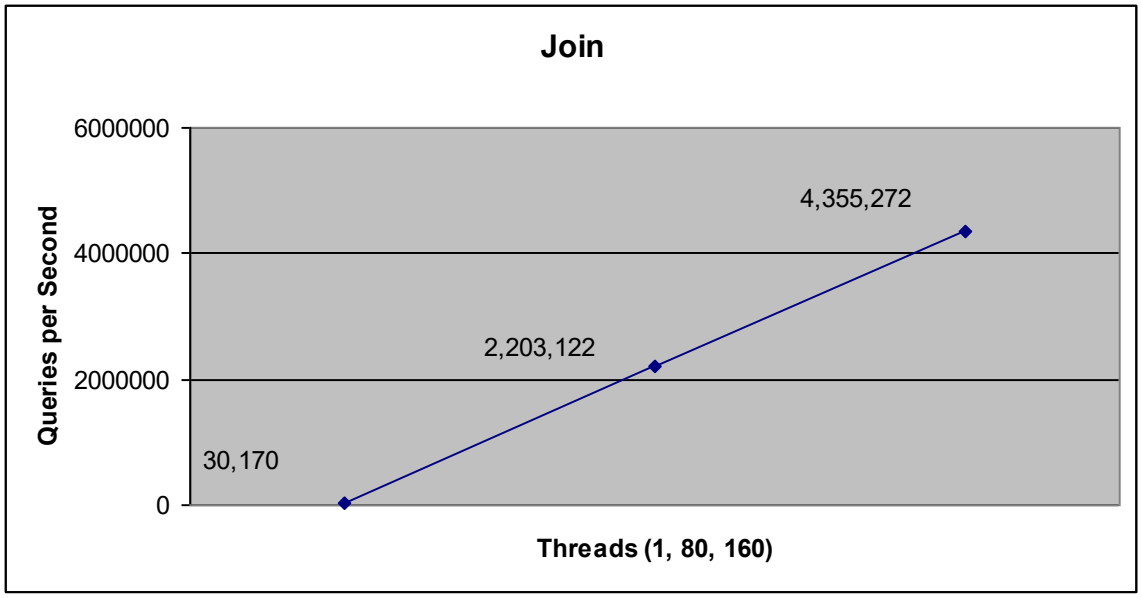

Chart 11 – JOIN performance of the SQL ODBC interface in queries-per-second.

The third and final test was a subquery:

```
SELECT * FROM persons
WHERE PERSON_ID IN
        (SELECT RECEIVER_ID FROM orders
        WHERE SENDER_ID IN
                (SELECT RECEIVER_ID FROM orders WHERE SENDER_ID = ?))
```

The code in the *eXtreme*DB native interface is shown in figure 6, and the SQL interface in figure 7.

```
int rawSubquery(mco_db_h pdb, mco_trans_h t, uint8 pID,
                struct STAT_DATA_R** pstat)
{
      mco_trans_h pt = t;
      persons P;
      uint8 rcv1, rcv2;
      orders T1, T2;
      mco_cursor_t cr1, cr2;
      int rss;
      uint4 rCnt=0;
      int rs = MCO_S_OK;

      if (!pt)
            if ((rs = mco_trans_start(pdb, MCO_READ_ONLY,
                        MCO_TRANS_FOREGROUND, &pt)) != MCO_S_OK)
            {
                  printf("Cant create transaction!\n");
                  return rs;
            }
      // searching for sender_id == % records... A
      if (orders_hkey3_index_cursor(pt, &cr1) == MCO_S_OK)
      {
            rs = orders_hkey3_search(pt, &cr1, MCO_EQ, pID);
            while(rs == MCO_S_OK)
            {
                  // load reciever_id && srch2
                  orders_from_cursor(pt, &cr1, &T1);
                  rs = orders_hkey3_compare(pt, &cr1, pID, &rss);
                  if(rs == MCO_S_OK && rss != 0)
                        break;
                  orders_RECEIVER_ID_get(&T1, &rcv1);
                  if(rs == MCO_S_OK)
                  {
                        if(orders_hkey3_index_cursor(pt, &cr2) ==
                            MCO_S_OK)
                        {
                              rs = orders_hkey3_search(pt, &cr2,
                                                MCO_EQ, rcv1);
                              while(rs == MCO_S_OK)
                              {
                                    // load receiver_id && hash srch
                                    orders_from_cursor(pt, &cr2, &T2);
                                    rs = orders_hkey3_compare(pt, &cr2,
                                                      rcv1, &rss);
                                    if (rs == MCO_S_OK && rss != 0)
                                          break;
                                    orders_RECEIVER_ID_get(&T2, &rcv2);
                                    if (rs == MCO_S_OK &&
                                        persons_hkey_find(pt, rcv2, &P)
                                        == MCO_S_OK)
                                          rCnt++;
                                    if (rs == MCO_S_OK )
                                          rs = mco_cursor_next(pt,
                                                            &cr2);
                              }
                        }
                        rs = mco_cursor_next(pt,&cr1);
```

```
                }
            }
        }
        if (!t)
                mco_trans_commit(pt);
        return rCnt;
}
```

Figure 7 – *eXtreme*DB native programming interface implementation of a SUBQUERY.

```
void doSubquery(int tid, void* pThrParam, mco_trans_h t, uint4 cnt)
{
    SQLHDBC hDbc = (SQLHDBC)pThrParam;
    SQLHSTMT hStmt;
    uint8 personId;
    SQLAllocStmt(hDbc, &hStmt); // allocate statement
    SQLPrepare(hStmt,
               (SQLCHAR*)"SELECT * FROM persons WHERE "\
" PERSON_ID IN (SELECT RECEIVER_ID FROM orders "\
" WHERE SENDER_ID IN (SELECT RECEIVER_ID FROM orders"\
" WHERE SENDER_ID=?) )",
                SQL_NTS)); // prepare query
    SQLBindParameter(hStmt, 1, SQL_PARAM_INPUT, SQL_C_UBIGINT,
                     SQL_BIGINT, 0, 0, &personId, 0, NULL);
    uint4 i;

    for (i = 0; i < cnt; i++)
    {
        personId = getPerson(tid);
        SQLExecute(hStmt); // execute prepared statement
        SQLCloseCursor(hStmt); // release result set

    }
    SQLFreeStmt(hStmt, SQL_DROP); // drop statement
}
```

Figure 8 – *eXtreme*DB SQL ODBC implementation of a SUBQUERY.

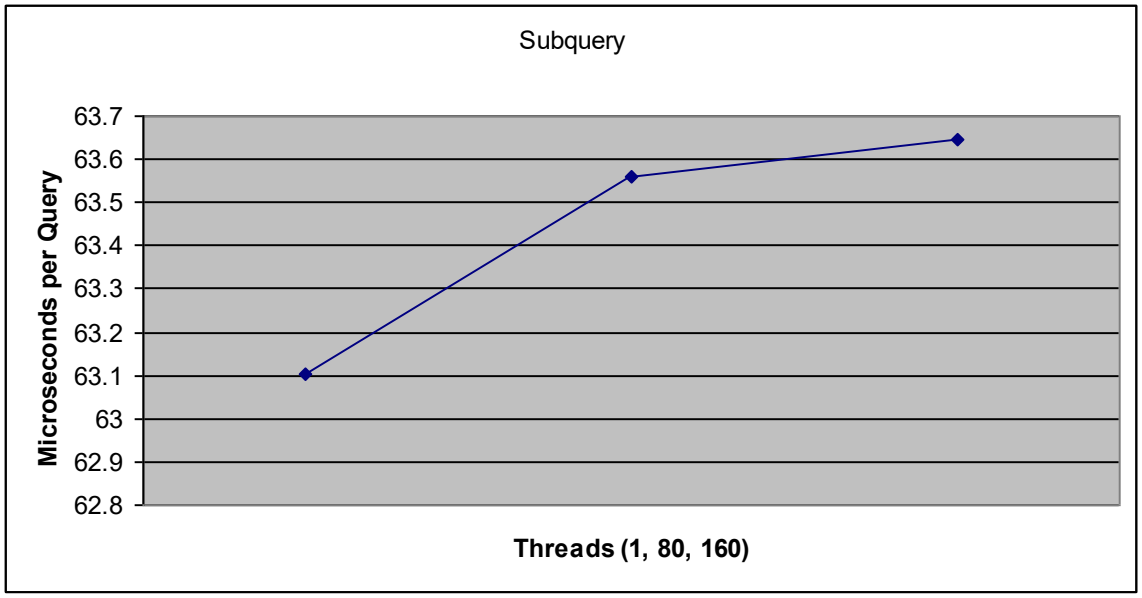

Chart 12 – SUBQUERY performance of the *eXtreme*DB native interface in microseconds-per-query.

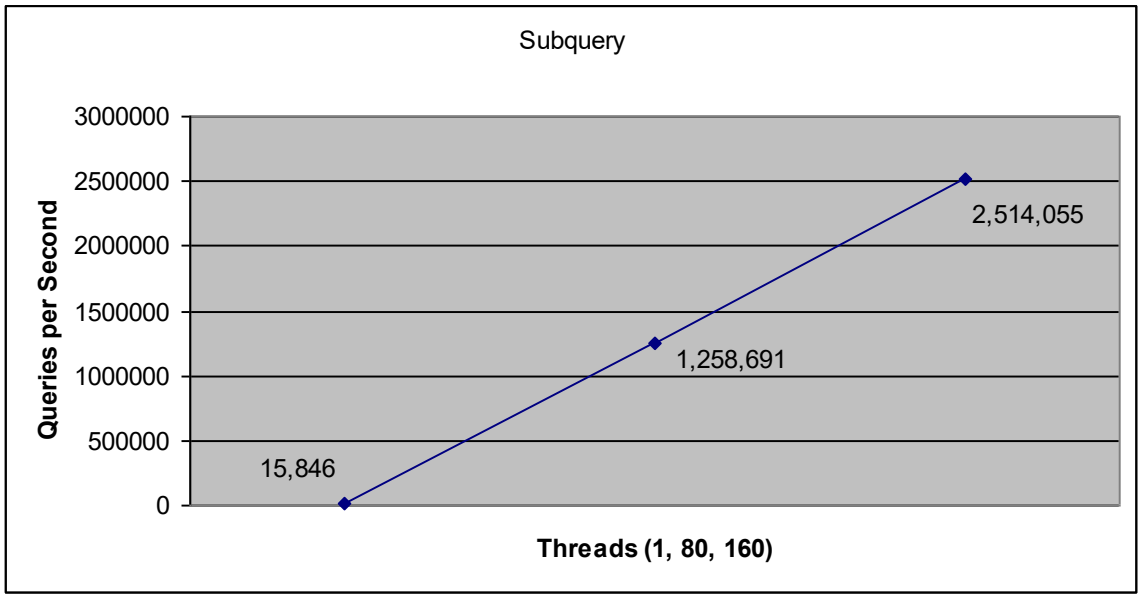

Chart 13 – SUBQUERY performance of the *eXtreme*DB native interface in queries-per-second.

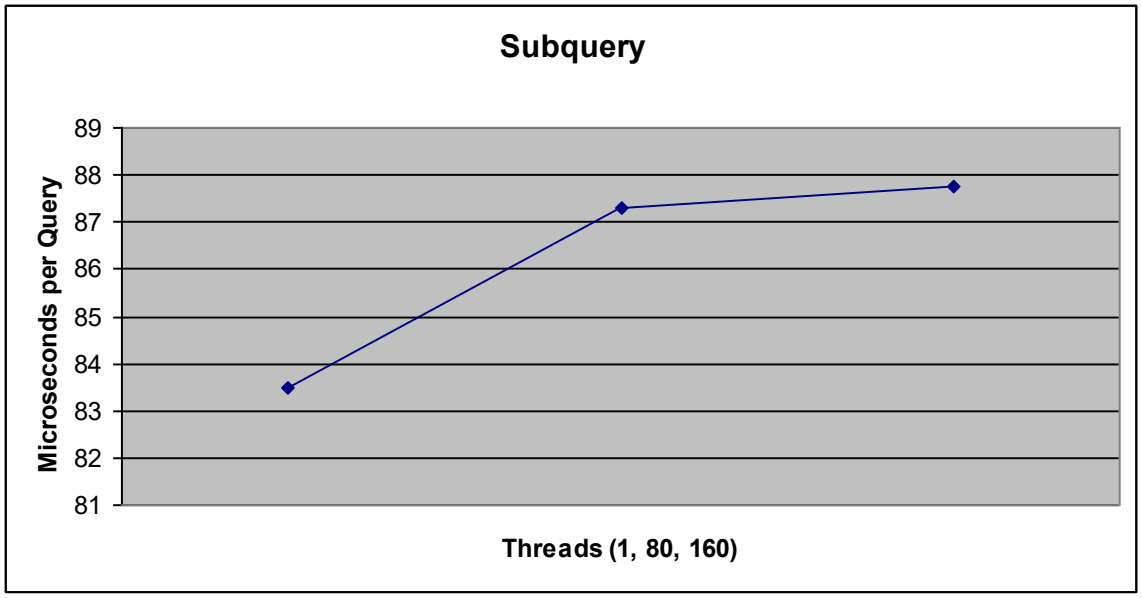

Chart 14 – SUBQUERY performance of the *eXtreme*DB SQL ODBC interface in microseconds-per-query.

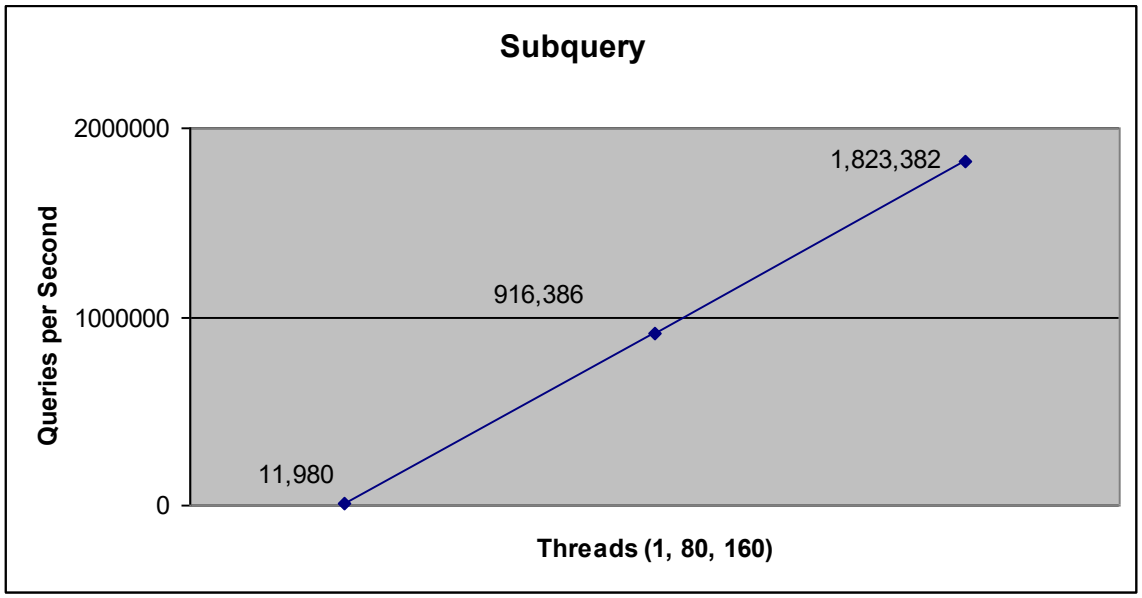

Chart 15 – SUBQUERY performance of the *eXtreme*DB SQL ODBC interface in queries-per-second.

Observations

Comparing the JOIN charts with the SELECT charts we can see that the SELECT charts show less performance improvement between 80 and 160 threads compared to the improvement between 1 and 80 threads. This a function of the random memory access inherent in the SELECT queries—where the test application conducts searches for

random values of a unique key that return a single row and how that specific test application's interacts with the NUMA architecture. Due to the random nature of the test, it is unlikely that a given row will be found in the local CPU cache, thus causing the cache line to be invalidated and reinitialized for any given query.

In contrast, the JOIN (and, later, SUBQUERY) tests exploit the "locality of reference" principle: when the database is provisioned, related rows are more likely to be placed on the same or adjacent pages so that the loop to fetch each matching row of the query is satisfied by data that was loaded into the CPU cache by the first row. Accordingly, there is more linear improvement in the total number of queries-per-second for the JOIN at 1, 80 and 160 threads, respectively. Put another way, the number of cache invalidations per row returned is far less, leading to a more uniform improvement in performance as the number of threads is scaled up.

Conclusion

The test results met expectations set by the 64-bit *eXtreme*DB's performance in tests involving somewhat smaller databases, and delivered truly groundbreaking results in the size of databases managed entirely in memory. The test proved that *eXtreme*DB can support an arbitrarily large number of concurrent processes/threads and deliver consistent performance. Performance did not degrade significantly with increased database size. The performance observed was in line with performance obtained with much smaller databases, given a processor clock speed of 1.6 Ghz.

The test proved that *eXtreme*DB can scale up to in-memory database sizes previously unexplored, and that the database load ("ingest") performance is largely unaffected by database size.

These results have important implications for organizations contemplating the next generation of high-performance applications that must process terabytes of data to guide high-level decisions, cutting-edge science or groundbreaking creative work. In-memory databases, while integrated in time-sensitive and embedded applications by many major corporations, are still viewed as somewhat exotic by many IT managers. The results of this benchmark test performed on off-the-shelf hardware and software indicate that:

- In-memory database size, measured either in terabytes or numbers of rows, can grow to well within the range of the largest corporate on-disk databases, with performance intact;
- Very large in-memory databases can be processed with outstanding performance using industry standard SQL or with the vendor's native application programming interface, and use of the native API provides a unique opportunity for optimization;

- For high-performance applications, an IMDS-based application can achieve:

    - QUERY performance of less than two microseconds per operation, or completion of more than 87.78 million queries per second;
    - JOIN performance under 14.5 microseconds per operation, or completion of more than 11.13 million joins per second.
    - SUBQUERY performance under 63.7 microseconds per operation, or more than 2.5 million subqueries per second

- Nearly linear scalability from a single processor core up to 160 processor cores can be achieved using the combination of McObject's 64-bit *eXtreme*DB in-memory database system and SGI's highly optimized hardware, with some minor and unavoidable drop-off in scalability occurring for operations that involve a higher proportion of system "housekeeping";

- IMDSs ingest data efficiently, with only a moderate decrease in performance as database size grows into the terabyte range;

- Backup and restore functions for this very large database required just a fraction of the time needed for initial provisioning, suggesting the persistence needs of time-sensitive, data-intensive applications can be met without undue latency.

The majority of organizations now employ "traditional" on-disk databases even for time-sensitive applications involving very large data stores. It is likely that on-disk data management will remain at the core of the installed base of traditional applications for the foreseeable future. But organizations are augmenting their IT infrastructure with new systems that exploit high performance hardware and software, in order to gain a competitive advantage. Many of these new systems, in fields ranging from business intelligence to pharmaceuticals research, are intended for tasks that are much more time-sensitive and data-intensive than older enterprise systems such as human resources and finance.

The results of this benchmark report indicate that IMDS technology meets the scalability requirements of this new category of application, and delivers dramatically faster query, join and subquery performance. Industry standard SQL is supported but a native API, which is intuitive to learn and use, delivers truly groundbreaking performance results. The IMDS is able to leverage the multi-core architectures becoming common in companies, university research labs, and government. With these capabilities and advantages well established, potential users of IMDSs have greater reason to move toward the use of the new technology, to secure their own advantage or to match the newly established performance standard when competitors deploy IMDS-based systems for time-sensitive, data-intensive applications.

---

[i] See http://www.wintercorp.com/VLDB/2005_TopTen_Survey/TopTenWinners_2005.asp. The Winter Corporation's survey examines results from disk-based databases. Applying such research to in-memory databases amounts to new terrain, and we believe the report you are now reading explores its furthest boundaries to date.

[ii] On-disk databases get slower as they get bigger because b-tree indexes get deeper. Each level in a b-tree tree equates to one logical disk I/O, so when a b-tree goes from 3 to 4 to 5 levels deep, the number of

I/Os to find a given value, either for a search or to find a value's position in the b-tree for an insert/update/delete, is greater. Thus, the bigger the database, the more average logical disk I/Os per index, and the slower the performance. In-Memory Database Systems also exhibit this (witness the 20% drop in ingest performance), but with much less of a performance drop-off. The difference between 3 and 5 memory operations compared to the difference between 3 and 5 disk operations is huge, so IMDSs incur far less overhead than on-disk databases as database size grows. Plus, the indexes in IMDSs are 'leaner' (no duplicated data), so the nodes don't fill up as fast and that keeps the index more shallow.