Altibase Challenges Oracle, IBM & Microsoft

Mature, battle-tested database is now open source

NEW YORK, Feb. 12, 2018 /PRNewswire/ -- On February 12 in 2018, Altibase, an enterprise grade relational database, announced that it is now open source.

"The database industry is going open source - the trend is clear," says Altibase Chairman, Paul Nahm. "But for discerning and prudent enterprise clients with mission critical applications, there is still one big question: Is there an open source database I can trust to be reliable 24/7? The answer is as of today, Yes, Altibase."
Altibase Challenges Oracle & Microsoft

Mature, battle-tested database is now open source.

NEW YORK, Feb. 12, 2018 /PRNewswire/ -- On February 12th, 2018, Altibase announced that it is now open source.

"The database industry is going open source - the trend is perfectly aligned with the needs of our customers and prudent enterprise clients with mission critical or high-volume transactional workloads. The question I get the most is 'What database can I trust to be reliable 24/7'? The answer is now clear: Altibase. With Altibase open source, our customers can now have both an enterprise-grade database platform and the system they can keep and expand as their business and data demands grow," said Haeng-Gu Lee, CEO of Altibase.

Altibase now joins an elite group of mission-critical open source database management systems, following in the footsteps of MySQL, PostgreSQL, and others. With Altibase open source, Altibase is positioned to better serve the needs of enterprise customers looking for a database platform that offers both reliability and performance.

TODAY’S AGENDA

Type Representation
In-Memory Data Layout
Storage Models
System Catalogs
DATA ORGANIZATION

**Fixed-Length Data Blocks**

- Index
- Block Id + Offset
- Fixed-Length Data Blocks

**Variable-Length Data Blocks**
DATA ORGANIZATION

One can think of an in-memory database as just a large array of bytes.
→ The schema tells the DBMS how to convert the bytes into the appropriate type.
→ Each tuple is prefixed with a header that contains its meta-data.

Storing tuples with as fixed-length data makes it easy to compute the starting point of any tuple.

Mapping virtual memory pages to database pages.
MEMORY PAGES

OS maps physical pages to virtual memory pages. The CPU's MMU maintains a TLB that contains the physical address of a virtual memory page.

→ The TLB resides in the CPU caches.
→ It can't obviously store every possible entry for a large memory machine.

When you allocate a block of memory, the allocator keeps that it aligned to page boundaries.
TRANSPARENT HUGE PAGES

Maintain larger pages automatically (2MB to 1GB)
→ Each page has to be a contiguous blocks of memory.
→ Greatly reduces the # of TLB entries

With THP, the OS will to reorganize pages in the background to keep things compact.
→ Split larger pages into smaller pages.
→ Combine smaller pages into larger pages.
→ Can cause the DBMS process to stall on memory access.

Almost every DBMS says to disable this feature:
→ Oracle, MemSQL, NuoDB, MongoDB, Sybase IQ

Source: Alexandr Nikitin
DATA REPRESENTATION

INTEGER/BIGINT/SMALLINT/TINYINT
→ C/C++ Representation

FLOAT/REAL vs. NUMERIC/DECIMAL
→ IEEE-754 Standard / Fixed-point Decimals

VARCHAR/VARBINARY/TEXT/BLOB
→ Pointer to other location if type is ≥64-bits
→ Header with length and address to next location (if segmented), followed by data bytes.

TIME/DATE/TIMESTAMP
→ 32/64-bit integer of (micro)seconds since Unix epoch
VARIABLE PRECISION NUMBERS

Inexact, variable-precision numeric type that uses the “native” C/C++ types.

Store directly as specified by IEEE-754.

Typically faster than arbitrary precision numbers.
→ Example: FLOAT, REAL/Doubles
VARIABLE PRECISION NUMBERS

#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %.20f\n", x+y);
    printf("0.3 = %.20f\n", 0.3);
}

Rounding Example

Output

x+y = 0.300000001192092895508
0.3 = 0.29999999999999998890
FIXED PRECISION NUMBERS

Numeric data types with arbitrary precision and scale. Used when round errors are unacceptable.
→ Example: **NUMERIC, DECIMAL**

Typically stored in a exact, variable-length binary representation with additional meta-data.
→ Like a **VARCHAR** but not stored as a string

Demo…
POSTGRES: NUMERIC

# of Digits
Weight of 1st Digit
Scale Factor
Positive/Negative/NaN
Digit Storage

typedef unsigned char NumericDigit;
typedef struct {
    int ndigits;
    int weight;
    int scale;
    int sign;
    NumericDigit *digits;
} numeric;
POSTGRES: NUMERIC

typedef unsigned char NumericDigit;

typedef struct {
    int ndigits;
    int weight;
    int scale;
    int sign;
    NumericDigit* digits;
} numeric;
typedef unsigned char NumericDigit;

typedef struct {
    int ndigits;
    int weight;
    int scale;
    int sign;
    NumericDigit *digits;
} numeric;

/* add_var() -
* Full version of add functionality on variable level (handling signs).
* result might point to one of the operands too without danger.
*/
int POSTGRESpnumeric_add(numeric *var1, numeric *var2, numeric *result)
{
    /* Decide on the signs of the two variables what to do */
    if (var1->sign == NUMERIC_POS)
    {
        if (var2->sign == NUMERIC_POS)
        {
            /* Both are positive result = +(ABS(var1) + ABS(var2)) */
            if (add_abs(var1, var2, result) != 0)
                return -1;
            result->sign = NUMERIC_POS;
        }
        else
        {
            /* var1 is positive, var2 is negative Must compare absolute values */
            switch (cmp_abs(var1, var2))
            {
                case 0:
                { /*
                    * ABS(var1) == ABS(var2)
                    * result = ZERO
                     */
                    zero_var(result);
                    result->rscale = Max(var1->rscale, var2->rscale);
                    result->dscale = Max(var1->dscale, var2->dscale);
                    break;
                }

                case 1:
                { /*
                    * ABS(var1) > ABS(var2)
                    * result = +(ABS(var1) - ABS(var2))
                     */
                    if (sub_abs(var1, var2) != 0)
                        return -1;
                    result->sign = NUMERIC_POS;
                    break;
                }

                case -1:
                { /*
                    * ABS(var1) < ABS(var2)
                    * result = -(ABS(var2) - ABS(var1))
                     */
                    break;
                }
            }
        }
    }
    else
    {
        /* var2 is positive, var1 is negative Must compare absolute values */
        switch (cmp_abs(var2, var1))
        {
            case 0:
            { /*
                * ABS(var1) == ABS(var2)
                * result = ZERO
                 */
                zero_var(result);
                result->rscale = Max(var1->rscale, var2->rscale);
                result->dscale = Max(var1->dscale, var2->dscale);
                break;
            }

            case 1:
            { /*
                * ABS(var1) > ABS(var2)
                * result = +(ABS(var1) - ABS(var2))
             */
                if (sub_abs(var1, var2) != 0)
                    return -1;
                result->sign = NUMERIC_POS;
                break;
            }

            case -1:
            { /*
                * ABS(var1) < ABS(var2)
                * result = -(ABS(var2) - ABS(var1))
               */
                break;
            }
        }
    }
    return 0;
}
CREATE TABLE AndySux (id INT PRIMARY KEY, value BIGINT);

char[]

<table>
<thead>
<tr>
<th>header</th>
<th>id</th>
<th>value</th>
</tr>
</thead>
</table>
CREATE TABLE AndySux (id INT PRIMARY KEY, value BIGINT);

char[]

header id value

reinterpret_cast<int32_t*>(address)
CREATE TABLE AndySux (value VARCHAR(1024));

INSERT INTO AndySux VALUES ("Andy has the worst hygiene that I have ever seen. I hate him so much.");
CREATE TABLE AndySux (value VARCHAR(1024));

INSERT INTO AndySux VALUES ("Andy has the worst hygiene that I have ever seen. I hate him so much.");

Variable-Length Data Blocks

Andy has the worst hygiene that I have ever seen. I hate him so much.

Andy | 64-BIT POINTER
NULL DATA TYPES

Choice #1: Special Values
→ Designate a value to represent NULL for a particular data type (e.g., INT32_MIN).

Choice #2: Null Column Bitmap Header
→ Store a bitmap in the tuple header that specifies what attributes are null.

Choice #3: Per Attribute Null Flag
→ Store a flag that marks that a value is null.
→ Have to use more space than just a single bit because this messes up with word alignment.
## NULL DATA TYPES

### Integer Numbers

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Size (Not Null)</th>
<th>Synonyms</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>2 bytes</td>
<td>1 byte</td>
<td>BOOLEAN</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BIT</td>
<td>9 bytes</td>
<td>8 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINYINT</td>
<td>2 bytes</td>
<td>1 byte</td>
<td></td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>4 bytes</td>
<td>2 bytes</td>
<td></td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>MEDIUMINT</td>
<td>4 bytes</td>
<td>3 bytes</td>
<td></td>
<td>-8388608</td>
<td>8388607</td>
</tr>
<tr>
<td>INT</td>
<td>8 bytes</td>
<td>4 bytes</td>
<td>INTEGER</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>BIGINT</td>
<td>12 bytes</td>
<td>8 bytes</td>
<td></td>
<td>-2 ** 63</td>
<td>(2 ** 63) - 1</td>
</tr>
</tbody>
</table>
The truth is that you only need to worry about word-alignment for cache lines (e.g., 64 bytes).

I’m going to show you the basic idea using 64-bit words since it’s easier to see...
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE AndySux (id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT);
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE AndySux (id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT);

```
<table>
<thead>
<tr>
<th>id</th>
<th>cdate</th>
<th>color</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
char[]
```

```
64-bit Word 64-bit Word 64-bit Word 64-bit Word
```

**32-bits**
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
CREATE TABLE AndySux (  
id INT PRIMARY KEY,  
cdate TIMESTAMP,  
color CHAR(2),  
zipcode INT  
);
```
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE AndySux (  
id INT PRIMARY KEY,  
cdate TIMESTAMP,  
color CHAR(2),  
zipcode INT  );

```sql
char[]
```
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE AndySux ( id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT );

char[]

id cdate c zipc

64-bit Word 64-bit Word 64-bit Word 64-bit Word
All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```sql
CREATE TABLE AndySux (  id INT PRIMARY KEY,  cdate TIMESTAMP,  color CHAR(2),  zipcode INT )
```
WORD-ALIGNED TUPLES

If the CPU fetches a 64-bit value that is not word-aligned, it has three choices:

→ Execute two reads to load the appropriate parts of the data word and reassemble them.
→ Read some unexpected combination of bytes assembled into a 64-bit word.
→ Throw an exception

Source: Levente Kurusa
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE AndySux ( id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT );
STORAGE MODELS

N-ary Storage Model (NSM)
Decomposition Storage Model (DSM)
Hybrid Storage Model
N-ARY STORAGE MODEL (NSM)

The DBMS stores all of the attributes for a single tuple contiguously.

Ideal for OLTP workloads where txns tend to operate only on an individual entity and insert-heavy workloads.

Use the tuple-at-a-time iterator model.
Choice #1: Heap-Organized Tables

→ Tuples are stored in blocks called a heap.
→ The heap does not necessarily define an order.

Choice #2: Index-Organized Tables

→ Tuples are stored in the primary key index itself.
→ Not quite the same as a clustered index.
N-ARY STORAGE MODEL (NSM)

Advantages
→ Fast inserts, updates, and deletes.
→ Good for queries that need the entire tuple.
→ Can use index-oriented physical storage.

Disadvantages
→ Not good for scanning large portions of the table and/or a subset of the attributes.
DECOMPOSITION STORAGE MODEL (DSM)

The DBMS stores a single attribute for all tuples contiguously in a block of data.
→ Sometimes also called *vertical partitioning*.

Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table’s attributes.

Use the vector-at-a-time iterator model.
DECOMPOSITION STORAGE MODEL (DSM)

1970s: Cantor DBMS
1980s: DSM Proposal
1990s: SybaseIQ (in-memory only)
2000s: Vertica, Vectorwise, MonetDB
2010s: “The Big Three”
       Cloudera Impala, Amazon Redshift, SAP HANA, MemSQL
TUPLE IDENTIFICATION

Choice #1: Fixed-length Offsets
→ Each value is the same length for an attribute.

Choice #2: Embedded Tuple Ids
→ Each value is stored with its tuple id in a column.

Offsets

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Embedded Ids

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
DECOMPOSITION STORAGE MODEL (DSM)

Advantages
→ Reduces the amount wasted work because the DBMS only reads the data that it needs.
→ Better compression.

Disadvantages
→ Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.
OBSERVATION

Data is “hot” when first entered into database
→ A newly inserted tuple is more likely to be updated again the near future.

As a tuple ages, it is updated less frequently.
→ At some point, a tuple is only accessed in read-only queries along with other tuples.

What if we want to use this data to make decisions that affect new txns?
BIFURCATED ENVIRONMENT

OLTP Data Silos

Extract Transform Load

OLAP Data Warehouse
HYBRID STORAGE MODEL

Single logical database instance that uses different storage models for hot and cold data.

Store new data in NSM for fast OLTP
Migrate data to DSM for more efficient OLAP
HYBRID STORAGE MODEL

Choice #1: Separate Execution Engines
→ Use separate execution engines that are optimized for either NSM or DSM databases.

Choice #2: Single, Flexible Architecture
→ Use single execution engine that is able to efficiently operate on both NSM and DSM databases.
Run separate “internal” DBMSs that each only operate on DSM or NSM data.
→ Need to combine query results from both engines to appear as a single logical database to the application.
→ Have to use a synchronization method (e.g., 2PC) if a txn spans execution engines.

Two approaches to do this:
→ Fractured Mirrors (Oracle, IBM)
→ Delta Store (SAP HANA)
FRACTURED MIRRORS

Store a second copy of the database in a DSM layout that is automatically updated.
→ All updates are first entered in NSM then eventually copied into DSM mirror.
Stage updates to the database in an NSM table. A background thread migrates updates from delta store and applies them to DSM data.
CATEGORIZING DATA

Choice #1: Manual Approach
→ DBA specifies what tables should be stored as DSM.

Choice #2: Off-line Approach
→ DBMS monitors access logs offline and then makes decision about what data to move to DSM.

Choice #3: On-line Approach
→ DBMS tracks access patterns at runtime and then makes decision about what data to move to DSM.
PELOTON ADAPTIVE STORAGE

Employ a single execution engine architecture that is able to operate on both NSM and DSM data.
→ Don’t need to store two copies of the database.
→ Don’t need to sync multiple database segments.

Note that a DBMS can still use the delta-store approach with this single-engine architecture.
**PELOTON ADAPTIVE STORAGE**

**Original Data**

```
UPDATE AndySux
SET A = 123,
    B = 456,
    C = 789
WHERE D = "xxx"

SELECT AVG(B)
FROM AndySux
WHERE C = "yyy"
```
### PELOTON ADAPTIVE STORAGE

#### Original Data

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hot</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cold</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**UPDATE** AndySux

**SET**
- **A** = 123,
- **B** = 456,
- **C** = 789

**WHERE**
- **D** = "xxx"

**SELECT** `AVG(B)`

**FROM** AndySux

**WHERE**
- **C** = "yyy"
PELOTON ADAPTIVE STORAGE

Original Data

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adapted Data

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UPDATE AndySux
SET A = 123,
B = 456,
C = 789
WHERE D = "xxx"

SELECT AVG(B)
FROM AndySux
WHERE C = "yyy"
TILE ARCHITECTURE

Introduce an indirection layer that abstracts the true layout of tuples from query operators.
TILE ARCHITECTURE

Introduce an indirection layer that abstracts the true layout of tuples from query operators.

<table>
<thead>
<tr>
<th>H</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tile Group Header

Tile #1

Tile #2

Tile #3

Tile #4
TILE ARCHITECTURE

Introduce an indirection layer that abstracts the true layout of tuples from query operators.

```
SELECT AVG(B) FROM AndySux WHERE C = "yyy"
```
TILE ARCHITECTURE

Introduce an indirection layer that abstracts the true layout of tuples from query operators.

```
SELECT AVG(B) FROM AndySux WHERE C = "yyy"
```
PELOTON ADAPTIVE STORAGE

- Row Layout
- Column Layout
- Adaptive Layout

Execution Time (ms)

<table>
<thead>
<tr>
<th>Sep-15</th>
<th>Sep-16</th>
<th>Sep-17</th>
<th>Sep-18</th>
<th>Sep-19</th>
<th>Sep-20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>Insert</td>
<td>Scan</td>
<td>Insert</td>
<td>Scan</td>
<td>Insert</td>
</tr>
<tr>
<td>1600</td>
<td>1200</td>
<td>800</td>
<td>400</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Sep-15: Initial execution time is high due to the new layout.
- Sep-16 to Sep-20: Gradual decrease in execution time as the system adapts.
PARTING THOUGHTS

A flexible architecture that supports a hybrid storage model is the next major trend in DBMSs → This will enable relational DBMSs to support all database workloads except for matrices in machine learning.