Four Things Every Developer (and DBA) Should Know about Oracle

Andrew V. Zitelli
Thales-Raytheon Systems
zitelli@raytheon.com

NoCOUG Fall Conference 2011 – November 9, 2011
Computer History Museum – Mountain View, California

Who Am I

- Bay Area native & UC Berkeley grad.
- Software developer for 35 years.
- Member of the OakTable Network.
  www.oaktable.net
- ACM member since 1974.
- Working with Oracle database products since 1992.
  - 10 other relational database products since 1982.
- With Thales-Raytheon Systems, Fullerton, CA since 2001.
  - Joint venture of aerospace firms Thales (France) and Raytheon (USA).
- Primarily working on new development of large multi-tier applications with complex Oracle databases.

Regarding Knowledge

Being ignorant is not so much a shame as being unwilling to learn. *Benjamin Franklin*

A person who won't read has no advantage over one who can't read. *Mark Twain*

An investment in knowledge pays the best interest. *Benjamin Franklin*

Contents

Introduction (1)

- Modern high performance software must typically support *high throughput* or *low response times*, or both.
- A key technology used to support high performance is concurrency, running multiple applications or application threads in parallel.
- One idea propounded by many developers is that software should be allowed to freely access data without any regard for how or where data is stored.
- *This overlooks the need to minimize contention and resource consumption systemwide, to support high levels of concurrency.*

Introduction (2)

- This presentation will discuss four key Oracle topics directly influencing contention, throughput and response times.
- *Although this presentation focuses on Oracle, many key points also apply to other database products.*
- Major topics include:
  - A consideration of transactions, commits and rollbacks.
  - Mistakes related to the use of unique identifiers in data.
  - Application use of Oracle's internal lock manager.
  - The importance of filtering data early.

Why Use Databases?

- *Proper use of databases should simplify the development and maintenance of application code.*
- Some of the key features databases support are:
  - Data integrity, assuring data contents are valid based on defined rules.
  - Data consistency, assuring data is synchronized in time.
  - Data security against damage, loss, and theft.
  - Managing the concurrent use of data by various users & applications.
  - Efficient execution of complex queries.
  - Deadlock detection and resolution.
  - Hiding new data requirements from existing code.

Oracle Instrumentation

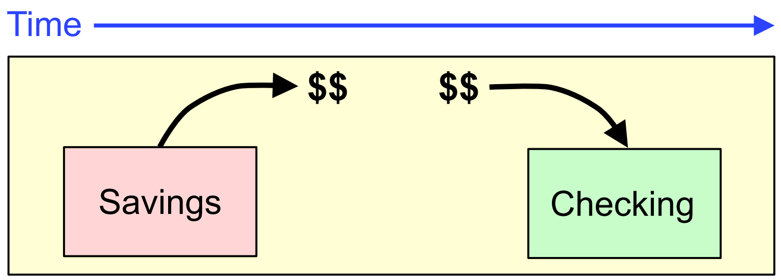
- This presentation makes reference to measurements collected using Oracle's *Extended SQL Trace Data,* also know as *10046 Trace Data.*
- This trace data is collected for individual Oracle processes, loosely equating to client connections (e.g., JDBC connections).
- Trace data contains details and elapsed times (in µs), regarding Oracle's internal activities and wait times for external activities.
- 10046 trace data records the exact sequence, timing and contents of SQL commands being received by the database, along with I/O details, network round trips and other pertinent details.
- *Many times developers assume they understand how their code interacts with Oracle but are wrong in their assumptions. 10046 trace data can be used to verify or refute one's assumptions.*

Topic #1: Commits, Rollbacks & Transactions

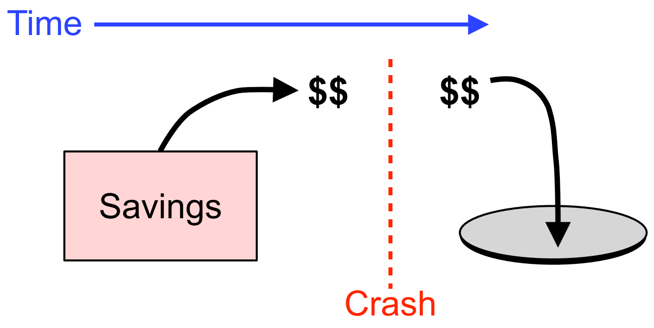
- *Excessive COMMITs and ROLLBACKs by applications are one of the leading causes of Oracle performance problems.*
- *Transactions* are one of the mechanisms Oracle uses to protect data integrity and consistency.
- *COMMIT* statements are used to make a given transaction's changes permanent.
- *ROLLBACK* statements are used to permanently remove all pending changes made within a transaction.
- Transactions assure that multiple related changes to data are guaranteed to be completed together, or not made at all.
- Transactions also assure one user cannot modify data in the process of being changed by another user.

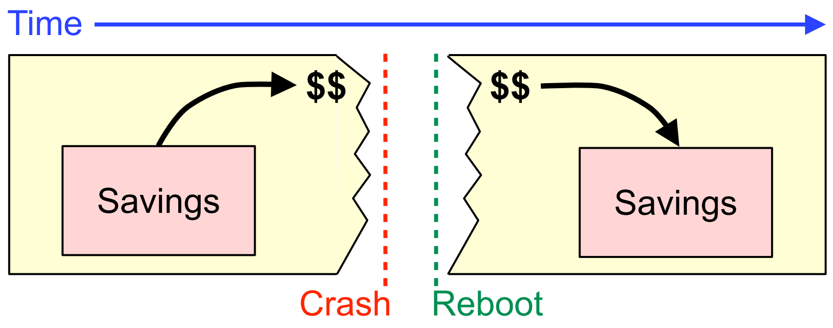
Transaction Example

- As an example, the actions of transferring money out of one account and into another account must always be completed together.



- *Without* transactions, data changes may be interrupted, leaving data in an inconsistent state.
- *With* transactions, any incomplete transactions are automatically rolled back following a failure.





COMMITs: Background Information (1)

- Oracle stores data (e.g., tables and indexes) inside "database blocks."
- Database blocks are permanently stored on disk.
- Blocks which are currently in use are also cached in memory.
- As data is modified and blocks are changed in memory, the modified blocks are not immediately written to disk.
- Instead, Oracle records sufficient information in "Redo Logs" to reconstruct the contents of every modified block, in the case of a database crash or failure.
- Modified blocks are eventually flushed to disk by one or more DBWR (Database Writer) background processes.

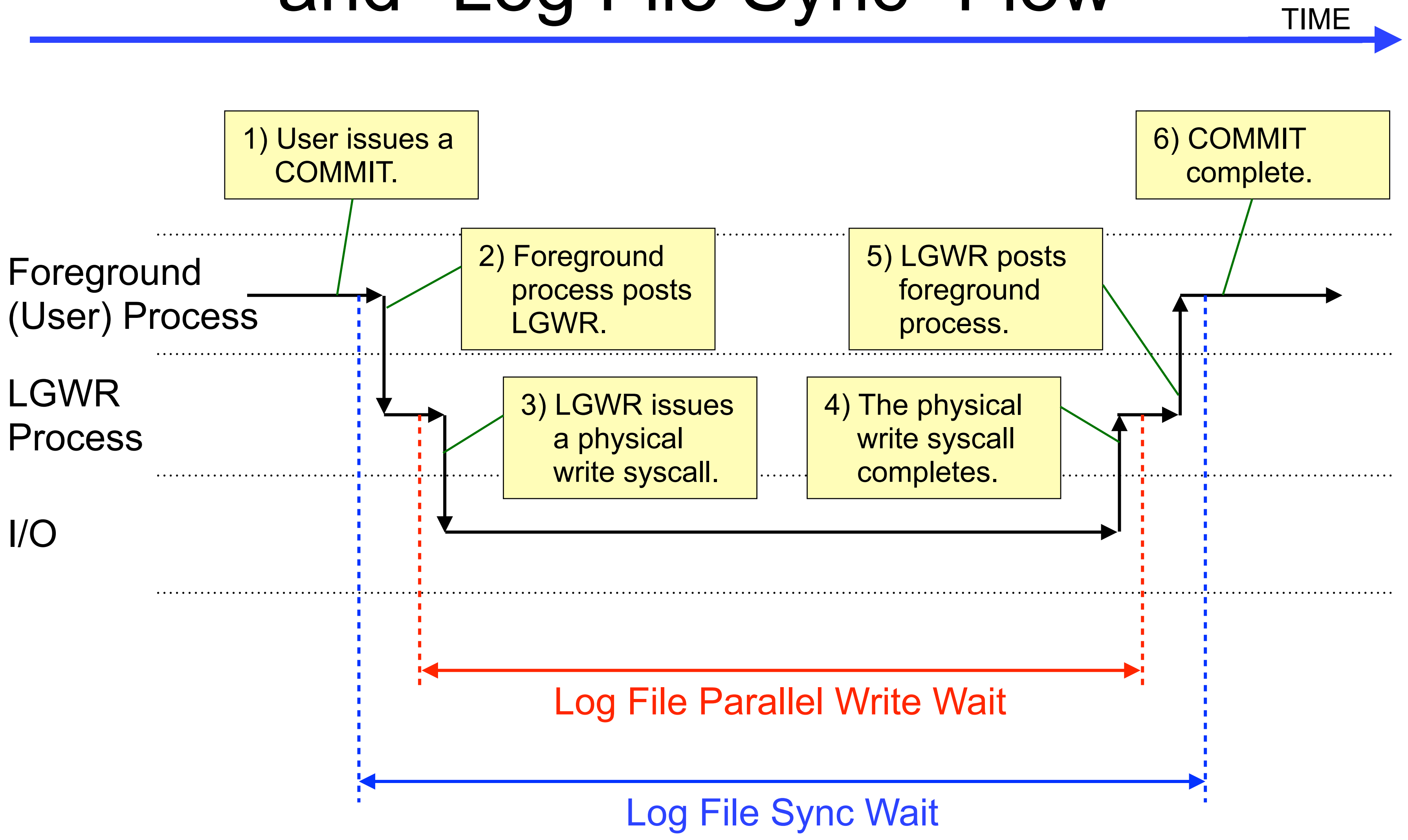
COMMITs: Background Information (2)

- When a COMMIT or ROLLBACK is performed, it is the corresponding Redo vectors, stored in memory, which are immediately flushed to the Redo Logs on disk.
- When this happens, the user's foreground process hands off control to a background process named LGWR (Log Writer) which is responsible for writing the buffered Redo to disk.
- The chart on the following page illustrates a simplified flow as a user process waits for LGWR to complete a physical write.
  - During a commit or rollback, 10046 trace data reports a foreground process's WAIT as a "Log File Sync," as shown below.

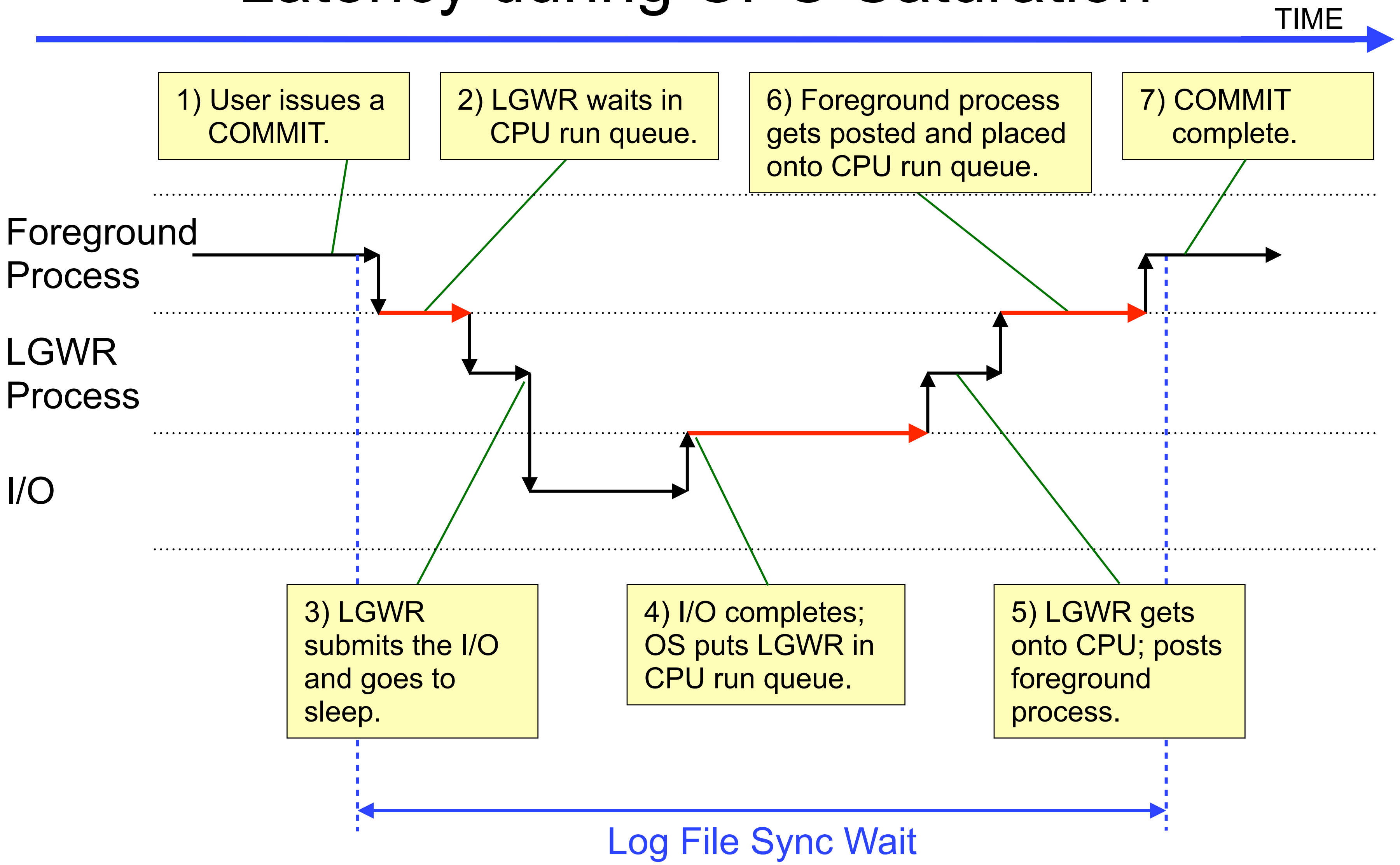
```
WAIT #7: nam='log file sync' ela=7680 buffer#=5081 sync scn=30860007 p3=0
   obj#=71617 tim=8345312529752
```

Elapsed time = 7680 microseconds

Idealistic Overview of COMMIT and "Log File Sync" Flow


TIME
1) User issues a COMMIT.
6) COMMIT complete.
Foreground (User) Process
2) Foreground process posts LGWR.
5) LGWR posts foreground process.
LGWR Process
3) LGWR issues a physical write syscall.
4) The physical write syscall completes.
I/O
Log File Parallel Write Wait
Log File Sync Wait

Log File Sync Performance: Scheduling Latency during CPU Saturation



TIME
1) User issues a COMMIT.
2) LGWR waits in CPU run queue.
6) Foreground process gets posted and placed onto CPU run queue.
7) COMMIT complete.
Foreground Process
LGWR Process
I/O
3) LGWR submits the I/O and goes to sleep.
4) I/O completes; OS puts LGWR in CPU run queue.
5) LGWR gets onto CPU; posts foreground process.
Log File Sync Wait

Measuring Commit Durations

- *Under load, duration of Commits and Rollbacks can vary widely.*
  - CPU saturation and I/O contention can contribute to very long *log file sync* times.
  - The following analysis of a 10046 trace file was generated using the tool *mrskew.* *Log file syncs* in the trace file averaged 8.49 ms, ranging from 295 µs to 720 ms.

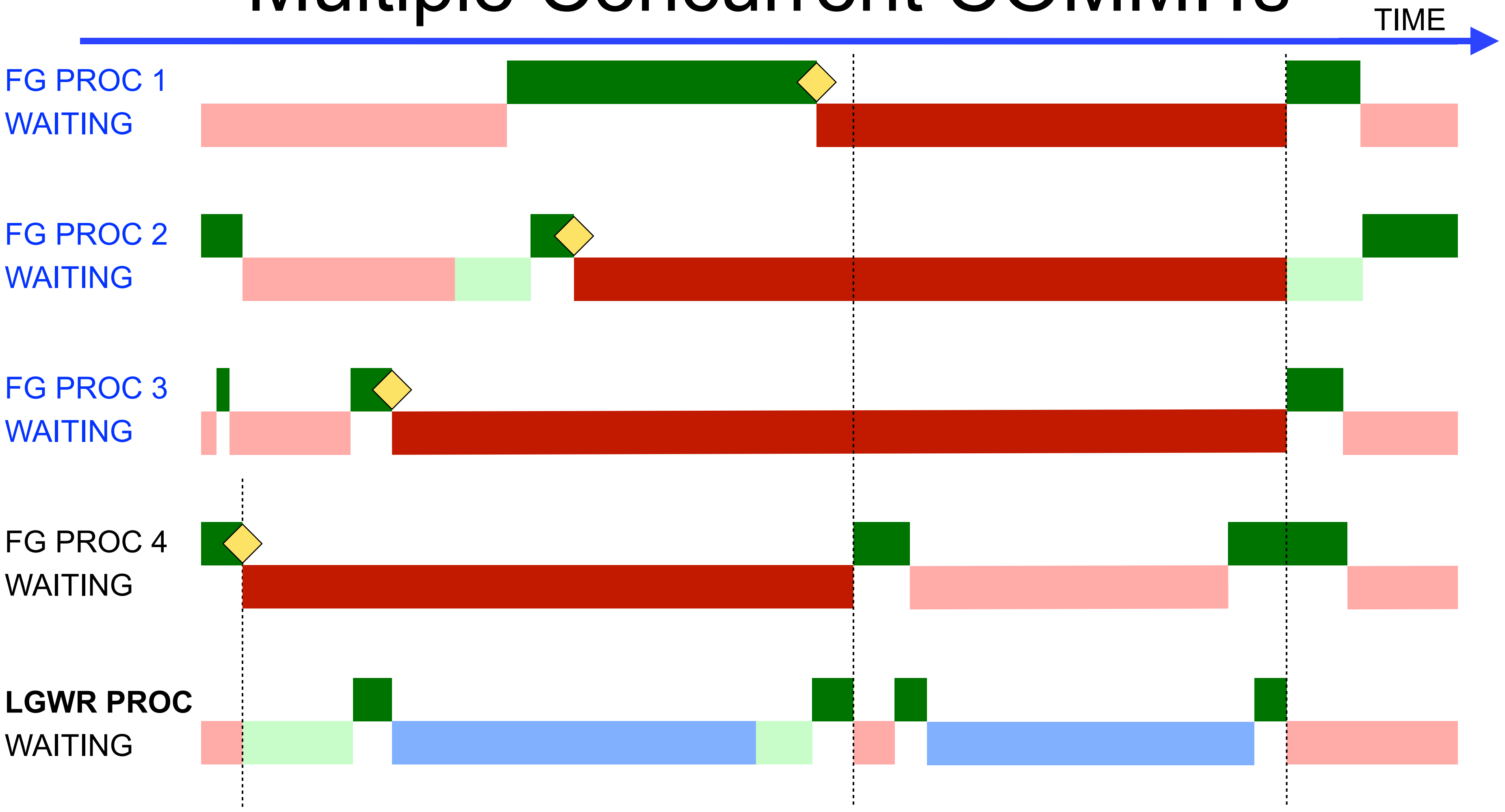
```
$ mrskew --nam='log file sync' ardbeg_ora_29335.trc

Matched event names:
        log file sync

Options: group   = ''
         name    = 'log file sync'
         where   = '1'

 RANGE {min <= e < max}                   DURATION         CALLS        MEAN         MIN         MAX
   0.000000    0.000001         0.000000    0.0%             0
   0.000001    0.000010         0.000000    0.0%             0
   0.000010    0.000100         0.000000    0.0%             0
   0.000100    0.001000         0.351132    1.6%           496    0.000708    0.000295    0.000998
   0.001000    0.010000         6.756450   31.1%          1987    0.003400    0.001000    0.009940
   0.010000    0.100000         0.847260    3.9%            42    0.020173    0.010035    0.077274
   0.100000    1.000000        13.754531   63.4%            32    0.429829    0.156393    0.719534
   1.000000   10.000000         0.000000    0.0%             0
  10.000000  100.000000         0.000000    0.0%             0
 100.000000 1000.000000         0.000000    0.0%             0
1000.000000    Infinity         0.000000    0.0%             0
                TOTAL (4)      21.709373  100.0%          2557    0.008490    0.000295    0.719534
```

2557 waits averaging 8.49 milliseconds

Behavior of Simultaneous COMMITs

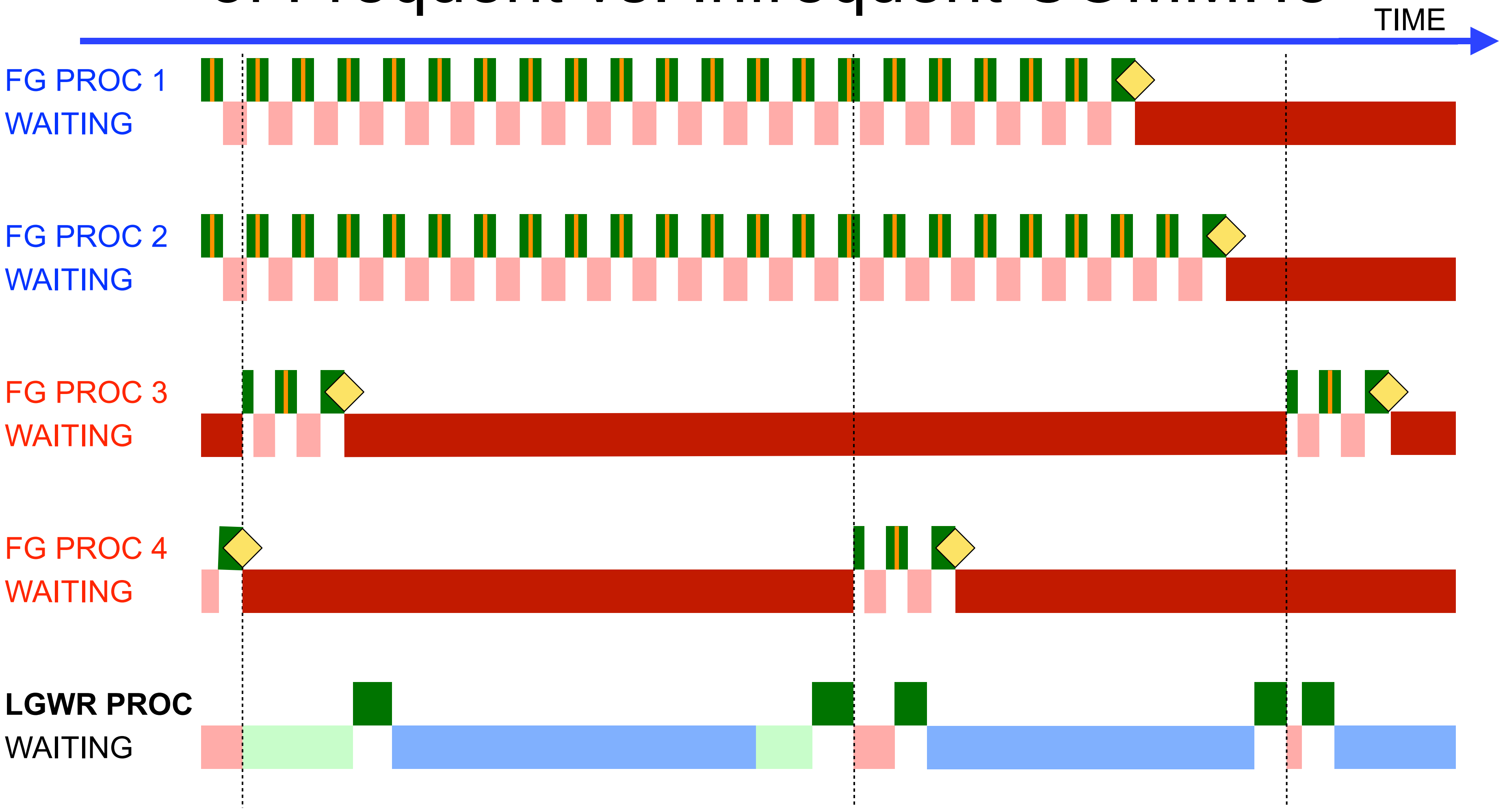
- If LGWR is already writing Redo when a COMMIT is executed, the user process must wait for LGWR's current write to complete, before LGWR can begin writing the user process's Redo.
- When multiple database sessions commit at the same time, the LGWR process flushes the Redo vectors for all pending commits and rollbacks.
- On the next slide, note that user FG (foreground) processes 1, 2 and 3 must all wait for completion of the Commit previously initiated by FG process 4.

Simplified Example of Multiple Concurrent COMMITs



KEY: ◆ COMMIT 🟩 Executing ▨ CPU Run Queue 🟥 Log File Sync Wait 🟦 I/O Waits ▨ Other Waits

Simplified Example Showing Throughput of Frequent vs. Infrequent COMMITs



KEY: ◆ COMMIT 🟩 Executing ▫ CPU Run Queue 🟥 Log File Sync Wait 🟦 I/O Waits ▪ Other Waits

🟩 Start of SQL Statement

Commit/Rollback Types & Sources (1)

- *The frequency of COMMITs which Oracle receives may be far different than developers expect.*
- Developers of high performance applications should assess the actual frequency of COMMITs emitted by their software.
- COMMITs can be initiated from a variety of places. These include but are not limited to:
  - COMMITs or ROLLBACKs explicitly executed by an application.
  - Enabled JDBC Autocommit feature (turned ON by default).
  - Frameworks like Hibernate and Enterprise Java Beans (EJBs).
  - PL/SQL procedures and functions executed by the client.
  - DDL commands like CREATE TABLE which force an implicit Commit.
  - Database triggers *never* perform any Commits or Rollbacks.
- *When COMMITs emanate from multiple sources, they are all executed by Oracle.*

Commit/Rollback Types & Sources (2)

- COMMIT and ROLLBACK both require a log file sync, unless the current transaction is READ-ONLY.
- The most detailed source for determining COMMIT frequency and durations is Oracle's 10046 trace data.
  - This may require tracing multiple database sessions simultaneously.
- Each XCTEND line records a transaction end. These are COMMITs or ROLLBACKs and include a Read-Only flag.

```
XCTEND rlbk=0, rd_only=0, tim=2291112322964  <- COMMIT
XCTEND rlbk=0, rd_only=1, tim=2291112475829  <- COMMIT, READ-ONLY
XCTEND rlbk=1, rd_only=0, tim=2291112639472  <- ROLLBACK
XCTEND rlbk=1, rd_only=1, tim=2291112928475  <- ROLLBACK, READ-ONLY
```

- Oracle's ASH and AWR utilities automatically collect summarized runtime data which can be helpful in assessing the past frequency and average durations of COMMITs.

Summary of Transactions, Commits and Rollbacks

- As stated earlier, excessive COMMITs and ROLLBACKs are one of the leading causes of Oracle performance problems.
- Use of transactions is critical to protecting data integrity, therefore commits needed to maintain data integrity should not be removed.
- Developers should attempt to identify and remove unnecessary commits. Poor application design may force more frequent commits than are genuinely necessary.
- *Developers of high throughput and low response time applications must not ignore the frequency of COMMITs being performed throughout their software stack.*

Regarding Database Agnostic Code (1)

- Commit frequency provides a good example of why "database agnostic" code often leads to poor application performance.
- Many database products support transactions but use significantly different mechanisms to implement them.
- Consider Oracle where Writers never block Readers.
  - This is supported by an architecture with fairly high Commit overhead.
  - *Oracle performance tends to improve as Commit frequency goes down.*
- In contrast, within many other databases, Writers do block Readers. Blocked readers wait for transactions to Commit.
  - Commits in these databases tend to incur a lower amount of overhead.
  - *In these databases, performance tends to improve as Commit frequency goes up.*

Regarding Database Agnostic Code (2)

- A recent web posting reads: *One of the main reasons I use Hibernate is that it provides the flexibility to switch to another database without having to rewrite any code.*
  - (1) Is flexibility to easily switch between databases an actual project requirement? Often not, and it comes with a very high price.
  - (2) How hard is it to actually port your application between databases? Vendors of popular databases provide migration tools in efforts to capture each other's business. Migration is often not difficult!
  - (3) Is this flexibility worth the mediocre performance incurred? You decide.
- *If developers choose to use a database agnostic approach, they risk poor throughput and response times on all databases.*
- *This can add to hardware costs which also increases software licensing costs related to the use of larger hardware.*

TOPIC #2: Unique Identifiers & Sequences (1)

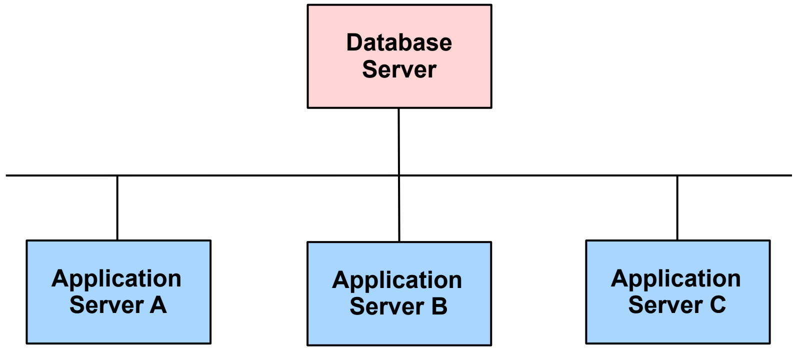
- Most applications rely on a unique identifier for every object being retrieved from or manipulated in a database.
- Rather than relying on *natural keys* found in data as identifiers, it is a common practice to use *surrogate keys*.
  - *Natural keys* are unique identifiers derived from data's natural contents. They may consist of one or more columns, of varying data types.
  - *Surrogate keys* are artificial identifiers added to provide unique ids which never change. These most often consist of a single numeric column, typically a positive integer.
- Surrogate keys can help simplify application code.
  - They normally consist of a single attribute which always uses the same datatype, across all object classes.
- In Oracle, *sequences* are a built-in mechanism for generating unique numbers, often used to generate surrogate key values.

Unique Identifiers & Sequences (2)

- *Three mistakes related to sequences and surrogate keys are commonly made by architects and developers.*
  - (1) They impose a requirement that unique key values be generated in exact chronological order.
  - (2) They impose a requirement that there be no gaps between key values.
  - (3) Their code makes an explicit call to Oracle for each key value being generated.
- Chronological Ordering and No Gaps increase contention between threads, as key values are being generated.
- Explicit calls to Oracle for every key value can incur significant delays and overhead on high throughput systems.

Sequences: Chronological Ordering (1)

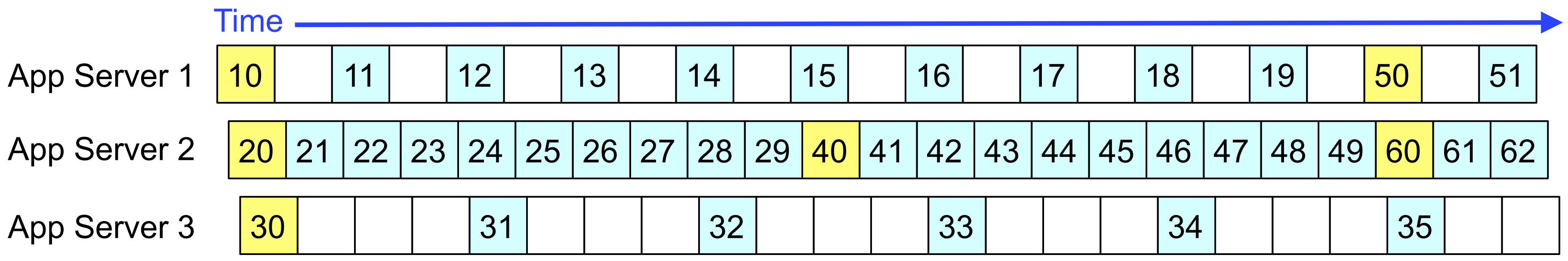
- *Required chronological ordering of surrogate key values increases contention between concurrent threads.*
- Consider the example of a high throughput system using application servers on 3 machines, with Oracle on a 4th.



- Chronological ordering precludes individual application servers from assigning their own key values locally.
- It requires a single source (Oracle or a designated app server) be used to dispense all values, increasing contention.
- This can require a network round trip to retrieve each key value.

Sequences: Chronological Ordering (2)

- One strong alternative is to have Oracle dispense values using an increment larger than 1, perhaps 100 or 1000.
- In this case, an application, using local single-threaded code, generates the unique key values between those values provided by Oracle.
- As applications utilize values at different rates, key values are not assigned in chronological order. Contention for unique keys is reduced.



Chronological Ordering: 10, 20, 30, 21, 11, 22, 23, 12, 24, 31, 25, 13, 26, 27, 14, 28, 32, 29, 15, 40, 41, . . .

🟨 Sequence Value from Oracle 🟦 Value Assigned Locally on each App Server

- Network round trips to retrieve values become much less frequent.
- This approach scales well as volume increases.

Sequences: Chronological Ordering (3)

- By default, Oracle sequences do not guarantee their values will be dispensed in the exact order requested.
- Oracle does provide an ORDERED option, which forces sequences to be dispensed in the exact order requested.
  - Under nominal loads, on single node databases, ordered and unordered sequences, have nearly the same response times.
  - Under heavy loads, CPU saturation can cause processes retrieving ordered sequence values to wait in the CPU run queue, subsequently blocking other processes also requesting values.
  - ORDERED sequences can cause serious contention issues when Oracle RAC (clustering) is in use.
- A better solution might be to use a timestamp for ordering.
  - A second column can be used as a tie-breaker, when needed.

Sequences: No Gaps (1)

- *A requirement to have No Gaps between key values can introduce serious response time and throughput issues.*
- *This requirement precludes the use of Oracle sequences because Oracle sequences never guarantee No Gaps.*
  - Once a sequence value has been dispensed, it cannot be put back.
  - If a transaction using a sequence value is rolled back, the sequence value is not reused.
  - If a sequence is defined to *cache* values in memory, and Oracle crashes, the unused cached values will never be used.
- The requirement for No Gaps forces all threads generating keys, to serialize when generating the gapless key values.
- Furthermore, each transaction assigning gapless key values must COMMIT or ROLLBACK, before any other transaction can be allowed to assign gapless key values.

Sequences: No Gaps (2)

- To assure no gaps are present when assigning new values, the existing high value must be read by each new transaction.
- Following is an example of logic required for multi-threading:
  (1) Allocate a lock to protect the critical section assigning the next key.
  (2) Read the current high value.
  (3) Insert data setting the new key value to the latest high value + 1, for each new row.
  (4) COMMIT the changes and release the lock.
- Simple tests of this gapless algorithm vs. gap prone sequences found the gapless version at least 30% slower.
  - The gapless version is prone to I/O, network and commit delays.
  - Commits like those profiled earlier could easily make the gapless algorithm tens or hundreds of times slower, raising contention issues.

Sequences: Extraneous Calls and Caching

- A poor but frequent practice is for applications to retrieve a sequence value, then use it in a subsequent SQL command.
- Where possible, the two statements should be combined into one, eliminating the extra network round trip and overhead.

```
insert into mytab values (my_seq.nextval, name, dob, . . .
```

- Internally, Oracle sequence high values are stored in a database table, to assure they survive reboots and crashes.
- Cache size in Oracle can be set separately for each sequence.
- Increasing the cache size above the default of 20 will make it more efficient, at the risk of losing more values during a crash.
- Setting cache size to NOCACHE does not remove the risk of gaps in sequence values.

Sequences: Index Risks from Ascending Values (1)

Use of sequence generated surrogate keys incurs at least two common risks associated with indexes used to enforce key uniqueness.

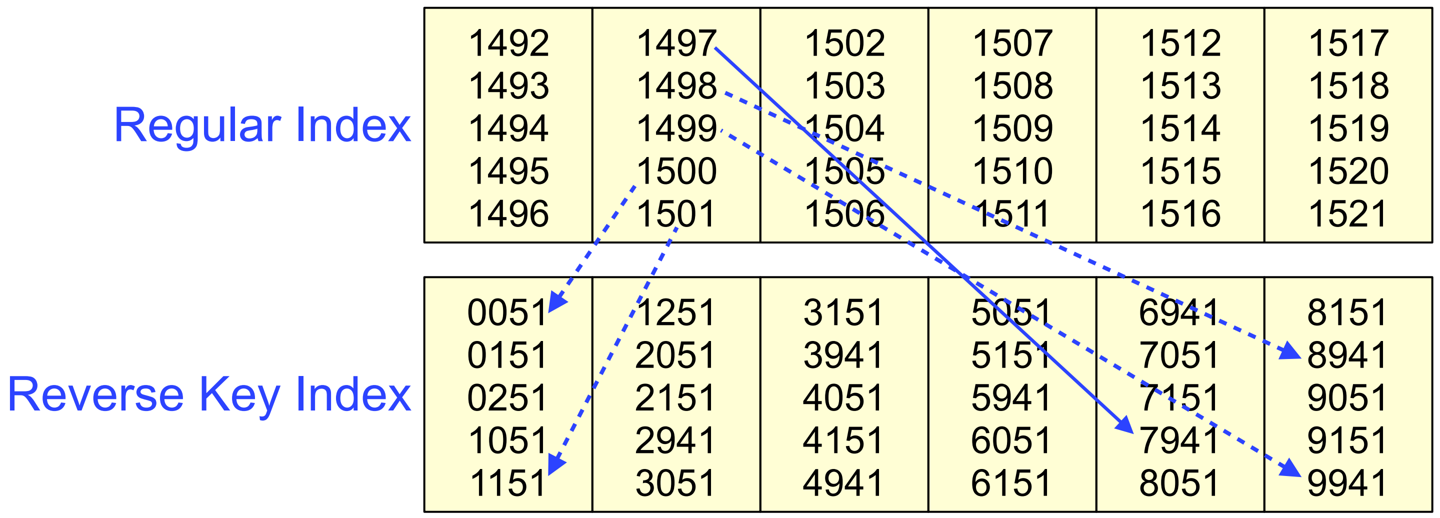
(1) On systems with large volumes of updates, high contention may occur for index blocks containing recent index entries.

(2) Depending on how a given table's data is updated and deleted over time, primary key indexes may perpetually grow in size.

- This occurs as an index grows from one end, while becoming sparsely populated on the other end (aka "right-handed indexes").
- Use of "reverse key" indexes can mitigate this problem under many circumstances.
- DBA's may need to periodically coalesce free space in these indexes.

Sequences: Index Risks from Ascending Values (2)

- Reverse key indexes sort index entries by inverting each column's value. For example "index" would be sorted as "xedni".
- This reduces potential hot blocks by spreading out sequential values over a wide range of index blocks.



Regular Index

1492	1497	1502	1507	1512	1517
1493	1498	1503	1508	1513	1518
1494	1499	1504	1509	1514	1519
1495	1500	1505	1510	1515	1520
1496	1501	1506	1511	1516	1521

Reverse Key Index

0051	1251	3151	5051	6941	8151
0151	2051	3941	5151	7051	8941
0251	2151	4051	5941	7151	9051
1051	2941	4151	6051	7941	9151
1151	3051	4941	6151	8051	9941

- Reverse key indexes can be used to retrieve individual rows but cannot be used to retrieve ranges of values:

```
SELECT * FROM MY_TAB WHERE ID = 1497;
SELECT * FROM MY_TAB WHERE ID BETWEEN 1497 AND 1501;
```

Sequences: Index Risks from Ascending Values (3)

- On high throughput systems with large tables, reverse key indexes may spread entires across so many index blocks, that index blocks tend to age out of memory between uses. This can result in excessive I/O.
- One solution is to replace sequence calls with calls to a function which manipulates the sequence values. For example:

```
CREATE OR REPLACE FUNCTION PID_NEXT_VALUE RETURN INTEGER IS
   PID INTEGER;
BEGIN
   PID := PID_SEQ.NEXTVAL;
   RETURN ((100000000000 * (MOD (PID * 37, 300) + 100)) + PID);
END PID_NEXT_VALUE;
/
```

- In this particular example, a value between 100e+11 and 399e+11, derived from the sequence value, is added to the sequence value. This algorithm provides 300 index insertion points. For example:

```
12345 => 16500000012345, 12346 => 20200000012346
```

- Function calls measure ~10% slower than calls to pid_seq.nextval.

Summary of Unique Identifiers and Sequences

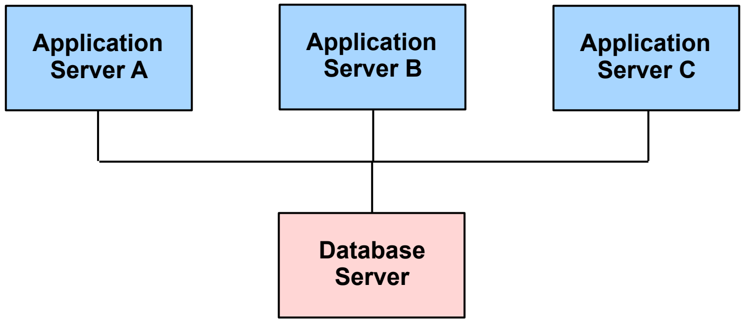
- Generally, unique identifiers in data should not be required to be in exact chronological order or gap free.
- When using sequences to generate surrogate key values, consider caching values in each application server. This can reduce network traffic, database overhead and contention for sequence values.
- Index values based on sequences may suffer from high contention or excessive size on disk. Reverse key indexes or some sort of function based key generation may help reduce these problems.

Topic #3: Critical Sections & DBMS_LOCK (1)

- Many applications contain sections of code which must never be executed by more than one thread at a time.
- In concurrent programming, these sections of code are referred to as *Critical Sections*.
- Critical sections normally require a locking mechanism to protect them, limiting access to one thread at a time.
- Oracle's DBMS_LOCK package provides an interface to Oracle's internal lock manager.
- This supports user defined locks for protecting critical sections and other critical resources, both inside and outside the database.

Critical Sections & DBMS_LOCK (2)

- Use of Oracle's DBMS_LOCK package is often preferable to implementing locks inside application code.
- Consider the following example where three application servers are in use, all executing the same application.



- When a given thread executes a critical section, it must first acquire a lock to prevent other threads from executing the same critical section, on any server.
- This requires a locking mechanism shared among application servers. DBMS_LOCK can be used for this.

Critical Sections & DBMS_LOCK (3)

- Some advantages of using DBMS_LOCK are:
  - The locks participate in Oracle's deadlock detection, along with locks on rows, tables and other Oracle resources.
  - Oracle 10046 trace data can be used to identify excessive lock contention.
  - It is easily called using JDBC stored procedure calls and other APIs.
  - Locks can be released prior to a Commit or Rollback (unlike SELECT .. FOR UPDATE).
  - Locks can be defined to automatically release on Commit or Rollback or to persist through a Commit or Rollback.
  - Locks are automatically released if the DB session is terminated.
  - DBMS_LOCK supports multiple lock types.
  - DBMS_LOCK's procedures can be called by database triggers.

Critical Sections & DBMS_LOCK (4)

- Examples of function/procedure calls from PL/SQL:

```
-- Attempt to acquire exclusive lock with ID #2100701. Give up after 60 sec.
LOCK_STATUS := DBMS_LOCK.REQUEST (ID => 2100701,
   LOCKMODE => DBMS_LOCK.X_MODE, TIMEOUT => 60, RELEASE_ON_COMMIT => TRUE);

-- Explicit release of user defined lock #2100701.
LOCK_STATUS := DBMS_LOCK.RELEASE (ID => 2100701);

-- Put the current session to sleep for the specified time in seconds.
DBMS_LOCK.SLEEP (SECONDS => 5);
```

- If a call to DBMS_LOCK.REQUEST needs to wait to acquire a lock, a line like the following will appear in the 10046 trace file:

```
WAIT #6: nam='enq: UL - contention' ela=5305032
   name|mode=1431044102 id=2100701 0=0 obj#=-1 tim=5439364938116
```

  – UL in the name 'enq: UL - contention' refers to User Lock.
  – ela= specifies the time waited in microseconds, 5.3 seconds above.
  – id= specifies the id of the lock requested.

Critical Sections & DBMS_LOCK (5)

- When 10046 trace files are available for analysis, they can be used to identify lock contention from calls to DBMS_LOCK.
- In the examples below, the *mrskew* analysis tool was used to look for UL lock contention, in a collection of trace files.

```
$ mrskew --name='enq: UL - contention' windmere_ora*.trc

 RANGE {min <= e < max}         DURATION      CALLS          MEAN          MIN          MAX
   0.000000     0.000001     0.000000    0.0%         0
   0.000001     0.000010     0.000000    0.0%         0
   0.000010     0.000100     0.000000    0.0%         0
   0.000100     0.001000     0.000467    0.0%         1      0.000467     0.000467     0.000467
   0.001000     0.010000     0.043560    0.0%         9      0.004840     0.001002     0.009146
   0.010000     0.100000     0.712841    0.2%        20      0.035642     0.010045     0.067194
   0.100000     1.000000     0.553350    0.1%         3      0.184450     0.144842     0.230371
   1.000000    10.000000     0.000000    0.0%         0
  10.000000   100.000000   414.682628   99.7%         7     59.240375    54.601764    60.016537
 100.000000  1000.000000     0.000000    0.0%         0
1000.000000     Infinity     0.000000    0.0%         0
                TOTAL (5)  415.992846  100.0%        40     10.399821     0.000467    60.016537

$  mrskew --name='enq: UL - contention' --group='$p2' --label='LOCK ID' windmere_ora*.trc

   LOCK ID              DURATION          CALLS          MEAN          MIN          MAX
   1000201        414.682628   99.7%          7     59.240375    54.601764    60.016537
   3300901          0.898332    0.2%         24      0.037430     0.000467     0.230371
   2100701          0.411886    0.1%          9      0.045765     0.032952     0.051456
 TOTAL (3)        415.992846  100.0%         40     10.399821     0.000467    60.016537
```

Critical Sections & DBMS_LOCK (6)

- *DBMS_LOCK provides a straightforward way to synchronize various applications and enforce serialization where needed.*
- The ability to explicitly release locks enables one to minimize the duration of critical sections without requiring a Commit or Rollback to release the locks.
  - DBMS_LOCK allows exception handlers to release locks.
- Execute privilege must be explicitly granted to DBMS_LOCK. This is not granted to every Oracle user by default.
- The DBMS_LOCK package is documented in the manual "Oracle Database PL/SQL Packages and Types Reference."

Topic #4: "Filter Early"

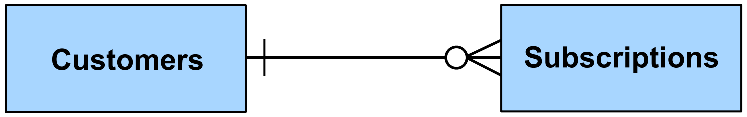
- Advice often repeated by SQL tuning experts is "Filter Early."
- When writing queries, this refers to the early exclusion from further consideration, of as much data as possible.
- Put differently: Make your system do as little work as possible.
- *This same advice applies on a larger scale when considering a system's complete software stack.*
- Many developers assume it makes little difference where data is filtered and manipulated, as long the work gets done.
- *For high performance systems, this assumption is wrong.*

Filter Early: Example (1)

- Consider the following two requests:
  1: Assemble a list of addresses for magazine subscribers in *Texas*, whose subscriptions expired during the past 90 days.

  2: Assemble a list of addresses for magazine subscribers in *California*, whose subscriptions expired during the past 90 days.

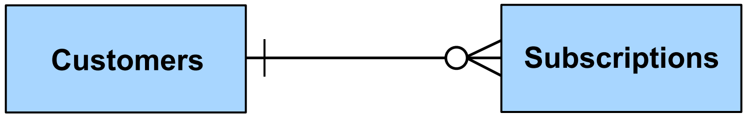
```
SQL> describe customers
Name                 Null?     Type
-------------------- --------- ------------
CUSTOMER_ID          NOT NULL  NUMBER(19)
CUSTOMER_NAME        NOT NULL  VARCHAR2(50)
STREET               NOT NULL  VARCHAR2(30)
CITY                 NOT NULL  VARCHAR2(30)
STATE                NOT NULL  VARCHAR2(2)
ZIP                  NOT NULL  NUMBER(5)
VERSION              NOT NULL  NUMBER(19)

SQL> describe subscriptions
Name                 Null?     Type
-------------------- --------- ------------
CUSTOMER_ID          NOT NULL  NUMBER(19)
SUBSCRIPTION#        NOT NULL  NUMBER(4)
EXPIRATION_DATE      NOT NULL  DATE
AUTO_RENEW                     VARCHAR2(1)
VERSION              NOT NULL  NUMBER(19)
```



Assume Oracle indexes exist on STATE and EXPIRATION_DATE and that they are used for the queries.

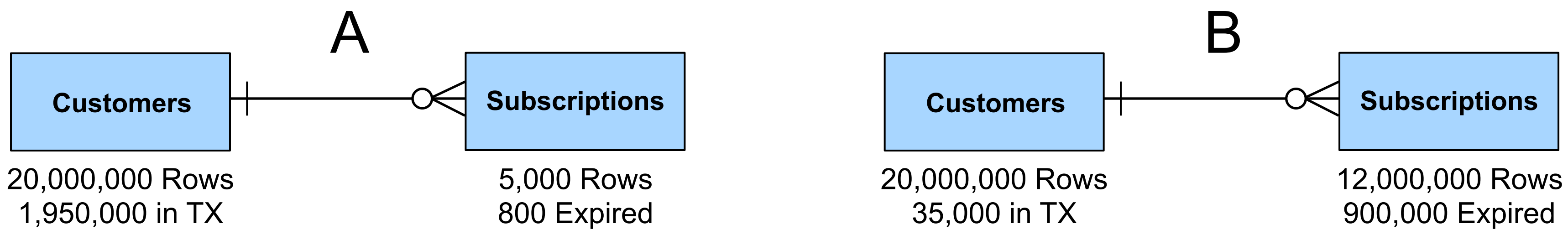
Filter Early: Example (2)

- *To answer these questions efficiently, where should one start, with the Customers table or the Subscriptions table?*
- The answer depends on the contents of the data in each table.



- CASE A:
  - Customers has 20,000,000 rows, with 1,950,000 in Texas and 2,100,000 in California.
  - Subscriptions has 5,000 total rows, with 800 expired in last 90 days.
- CASE B:
  - Customers has 20,000,000 rows, with 35,000 in Texas and 2,100,000 in California.
  - Subscriptions has 12,000,000 rows, 900,000 expired in last 90 days.

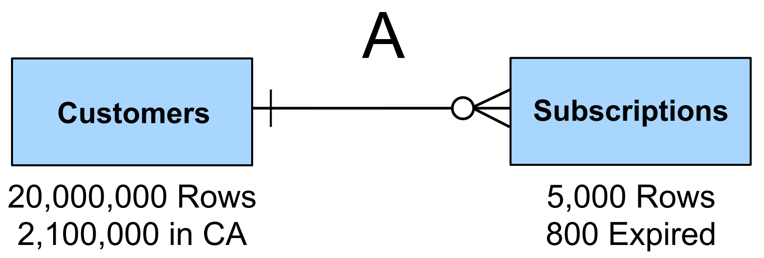
Filter Early: Example *Texas* Subscribers

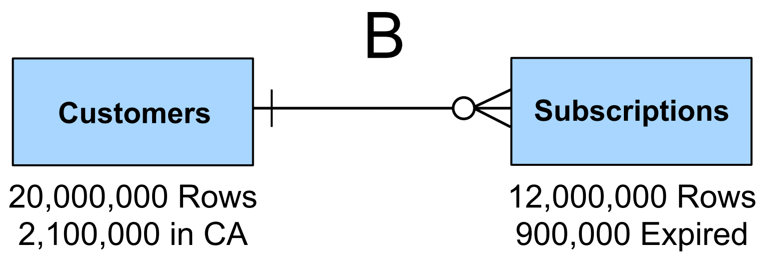


A:
- Starting with Customers, one must read 1,950,000 rows for Texas and then probe Subscriptions 1,950,000 times looking for matches.
- Starting with Subscriptions, one must read 800 rows and then probe Customers 800 times to retrieve addresses.
- *Subscriptions* is the better place to start (~1,600 vs. 3,900,000).

B:
- Starting with Customers, one must read 35,000 rows and then probe Subscriptions 35,000 times looking for matches.
- Starting with Subscriptions, one must read 900,000 rows and then probe Customers 900,000 times to retrieve addresses.
- *Customers* is the better place to start (~70,000 vs. 1,800,000).

Filter Early: Example *California* Subscribers





A:
- Starting with Customers, one must read 2,100,000 rows and then probe Subscriptions 2,100,000 times looking for matches.
- Starting with Subscriptions, one must read 800 rows and then probe Customers 800 times to retrieve addresses.
- *Subscriptions* is the better place to start (~1,600 vs. 4,200,000).

B:
- Starting with Customers, one must read 2,100,000 rows and then probe Subscriptions 2,100,000 times looking for matches.
- Starting with Subscriptions, one must read 900,000 rows and then probe Customers 900,000 times to retrieve addresses.
- Unlike Texas, *Subscriptions* is the best place to start (~1,800,000 vs. 4,200,000).

Filter Early: Summary

- Oracle collects & maintains a wide variety of data statistics.
- *Middle tier frameworks and applications do not have the benefit of up-to-date data statistics.*
- Oracle's data statistics help its query optimizer determine an efficient execution plan (retrieval strategy) for each query.
- Inefficient execution plans are often thousands of times slower, than efficient plans returning exactly the same data.
- Filtering data in the middle tier incurs a high risk of inefficient retrieval strategies which increase resource consumption across many hardware and software components in a system.
- *Applications should filter as much data as possible in the database, retrieving only that data which they require for their core business logic.*

Conclusion (1)

- Large software systems increasingly rely on multi-threading and high concurrency to achieve high performance.
- Oracle, like other database products, includes a wide variety of features which leverage its architecture to support high concurrency, high throughput and rapid response times.
- Systems designed to remain database neutral, neglecting to use database specific enhancements, run strong risks of high contention, excessive overhead and poor performance.
- These systems tend to require disproportionately large amounts of hardware for the tasks at hand, also leading to increased software licensing costs.

Conclusion (2)

- Developers and architects need to understand their target databases, learning to use their features efficiently in support of the systems they are developing.
- Filtering data in the database as much as possible, instead of inefficiently filtering in other software tiers, is another critical consideration when developing high performance software.
- When used properly, databases should simplify software development, reduce maintenance costs and promote high system performance.

Q & A

References & Further Reading

Millsap, C. 2011. *Mastering Performance with Oracle Extended SQL Trace.* Method-R Corporation. *http://www.method-r.com/downloads/doc_download/72-mastering-performance-with-extended-sql-trace*

Millsap, C.; Holt, J. 2003. *Optimizing Oracle Performance.* O'Reilly. ISBN 059600527X.
This book is highly recommended to all DBAs and developers.
Chapters 1 to 4 are strongly recommended for all developers.

Oracle Corporation. 2010. *Interpreting Raw SQL_TRACE and DBMS_SUPPORT.START_TRACE Output - Note 39817.1.*

Oracle Corporation. 2010. *Oracle Database PL/SQL Packages and Types Reference, 11g Release 2 (11.2).* Part Number E16760-05.

Oracle Corporation. 2010. *Oracle Database Reference, 11g Release 2 (11.2).* Part Number E17110-05.

Põder, T. 2010. *Understanding LGWR, Log File Sync Waits and Commit Performance.* *http://files.e2sn.com/slides/Tanel_Poder_log_file_sync.pdf*

My Primary Tools for Trace File Analysis

- MR Tools, a collection of powerful command line tools for 10046 trace file analysis. One of these tools, *mrskew*, is shown in some of the examples. Available from Method-R Corporation at:
  http://www.method-r.com/software/mrtools
- Method-R Profiler, also available from Method-R Corporation at:
  http://www.method-r.com/software/profiler
- Personal tools I have written in PL/SQL and Perl. *Several of these are available upon request writing me at: zitelli@raytheon.com*
- A fast text editor adept at handling files exceeding 100 Mb with potentially thousands of characters per line. I prefer BBEdit (Mac) and TextPad (PC).
- Simple UNIX commands and utilities like grep, awk, sort, wc, and perl.
- Oracle documentation and web searches.
- Write me if you have questions or need further assistance.