



Oracle database scalability

John Kanagaraj, MTS2, DB Engineering

November, 2013



About the presenter

- John Kanagaraj
 - Member of Technical Staff in Database Engineering team @ PayPal
 - Frequent speaker at Oracle OpenWorld, NoCOUG and IOUG COLLABORATE conferences
 - Oracle ACE, author, IOUG SELECT Journal Editor
 - Contact me on LinkedIn (search for my name) or email at john.kanagaraj@gmail.com



Agenda

- A definition of scalability
- Components of Scalability
- Breaking it down
 - Oracle specific capabilities/optimizations
 - Hardware specific capabilities
- Scalability Anti-patterns
- Wrap up, Q & A
- (Run this PPT in Slide Show mode if reviewing this later, and optionally read more information in the notes section)



Scalability - a definition

- Scalability - Ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth
- By extension.... Database scalability is the ability of an Oracle database based data storage and retrieval system (and all it encompasses in terms of CPU, memory, storage and Oracle/ OS components, as well as the database objects involved) to expand and accommodate the growth that is demanded of it
- Lots of words, but essentially boils down to two:
 - Capacity
 - Capability
- Two directions you can go: “Horizontal” and “Vertical”

Confidential and Proprietary

Vertical scaling, also known as “scale up” is usually achieved by buying hardware that typically has faster CPU’s, and by increasing the *capability* of the other layers in the system. Horizontal scaling, also known as “scale out” is usually achieved by moving from a single node to a multi-node configuration or by adding nodes in an already clustered configuration, essentially adding to the *capacity* of the system. Both of them have their pros and cons, and both address slightly different scalability requirements

Why bother about Scalability?

- Healthy Businesses are always growing and expanding
 - New users (more users performing more of the same actions)
 - New products (new capabilities in the application/database)
 - New markets (usually a combination of the two, may be geo-separated)
 - Organic traffic increases
 - Seasonal changes
- Work vs. Database Load/Usage
 - Should normally be linear
 - In practice, not proportional
 - Additional load usually exposes hidden weaknesses, limits
 - Disastrous if “tipping point” is reached, usually at the wrong time ☹



The fallout and the response

- Guess who gets the blame (hint: you know them well)
- The response:
 - Mad scramble, endless meetings and hand-wringing
 - Followed by the blame game
 - “Go fix it” mandate
 - Throw more hardware (in a hurry) at the problem
 - Sometimes this makes it worse (we will look at some examples)
- The correct approach:
 - Understand “scalability inhibitors” + Measure, Monitor, Move proactively
 - Good capacity planning – up and down the stack (not just the DB!)
 - Scale up and Scale out (vertical and horizontal scale-outs) proactively





A common scalability inhibitor

- Heap table with one or more “right growing” indexes
 - Primary Key: Unique index on a NUMBER column
 - Key value generated from an Oracle Sequence (NEXTVAL = 1)
 - I.e. “monotonically” increasing ID value
 - High rate of insert (> 5000 inserts/second) from multiple sessions
 - Multiple indexes, typically leading date/time series or mono-valued
 - E.g. Oracle E-Business Suite’s FND_CONCURRENT_REQUESTS
- Here’s the Problem:
 - All INSERTing sessions need one particular index block in CURRENT mode (as well as one particular data block in CURRENT mode)
 - Question: Would you use RAC to scale out this particular workload?



A quick deep dive

- Here's what happens to accommodate the INSERT
 - Assume the current value of the PK is 100, and NEXTVAL = 1
 - Assume we have 'N' sessions simultaneously inserting into that table
 - Session 1 needs to update the Index block (add the Index entry for 100)
 - Session 2 wants the same block in CURRent mode (add another entry for 101; needs the same block because the entry fits in the same block)
 - Session 3... N also want the same block in CURRent mode at the very same time (as all sessions will have "nearby" values for index entry)
 - Block level pins/unpins (+ lots of other work – Redo/Undo) required....
 - Same memory location (SGA buffer for Index block) accessed
 - Smaller but still impacting work for buffer for Data block
 - **Rate of work constrained by CPU speed and RAM access speeds**

Confidential and Proprietary

It is necessary to understand CURRent block access versus CR (Consistent Read) access. To make a change in a block (Index block) in this case, the session making the change has to obtain the "CURRent" version of the block (represented as a CURR get) - and there can be only _one_ version of the block in the entire cluster - the locks/pins and other concurrency requirements necessary to make this happen has a lot of overhead and essentially, _this_ concurrency requirement essentially single threads the operation. This is then considered as placing an upper limit on the scalability of this operation - if this is important enough, then this particular operation can bottleneck the entire application.

Essentially, this translates to how quickly a CPU can update a single memory location, and faster CPU's and faster Memory access can "speed up" access and increase the scalability runway.



A quick deep dive

- What if you use RAC to “scale out” this workload?
 - Assume “N” sessions simultaneously inserting from 2 RAC nodes (2xN)
 - In addition to previously described work, you need to
 - Obtain the Index block from remote node in CURRent mode
 - Session 1 (Node 1) updates Index block with value 100
 - Session 2 (Node 2) requests block in CURRent mode (value 101)
 - LMS processes on both nodes churn CPU co-ordinating messages and block transfers back and forth on the interconnect
 - Flush redo changes to disk on Node 1 before shipping CURRent block to Node 2 (*gated by RedoWriter response!!!*)
 - Sessions block on “gc **current** <state>” waits during this process
 - CPU, Redo IO, Interconnect, LMS/LMD processes involved

Confidential and Proprietary

The fact that CPU, the latency of Redo IO, the Interconnect overheads as well as the ability of the LMS/LMD processes to schedule themselves on to CPU adds to the concurrency needs when RAC is involved in changing a hit block brought about by this pattern. In other words, trying to scale out a workload such as this may not really work....



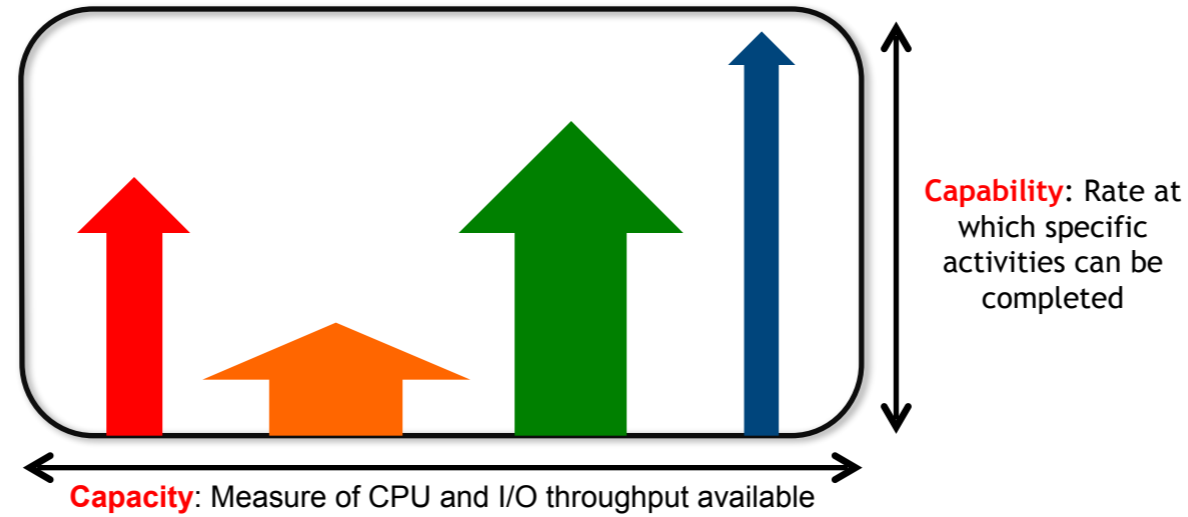
A quick deep dive

- Some solutions
 - Spread the pain for the right growing index
 - Use Reverse Indexes (cons: Range scan not possible)
 - Use Hash partitioned indexes (cons: All partitions probed for Range scan, Need Partitioning Option, Additional administration)
 - Prefix RAC node # (or some identifier per node) to key
 - Use a modified key: Use Java UUID, Other distinct prefix/suffixes
 - Use Range-Hash Partitioned tables with Time based ID as key
 - E.g. Epoch Time (# of seconds from Jan 1, 1970) + Sequence value for lower bits
 - Enables Date/Time based partitioning key
 - Unique values allow Local Index to be unique

Scalability patterns



Different components in a Database Workload – Single node Database server



- Single database server hosting four different, distinct workloads
- Each workload is (ultimately) a set of CRUD operations on tables/indexes
- May be distinctly segregated by many dimensions (schema, application, mid-tier, etc.)

Confidential and Proprietary

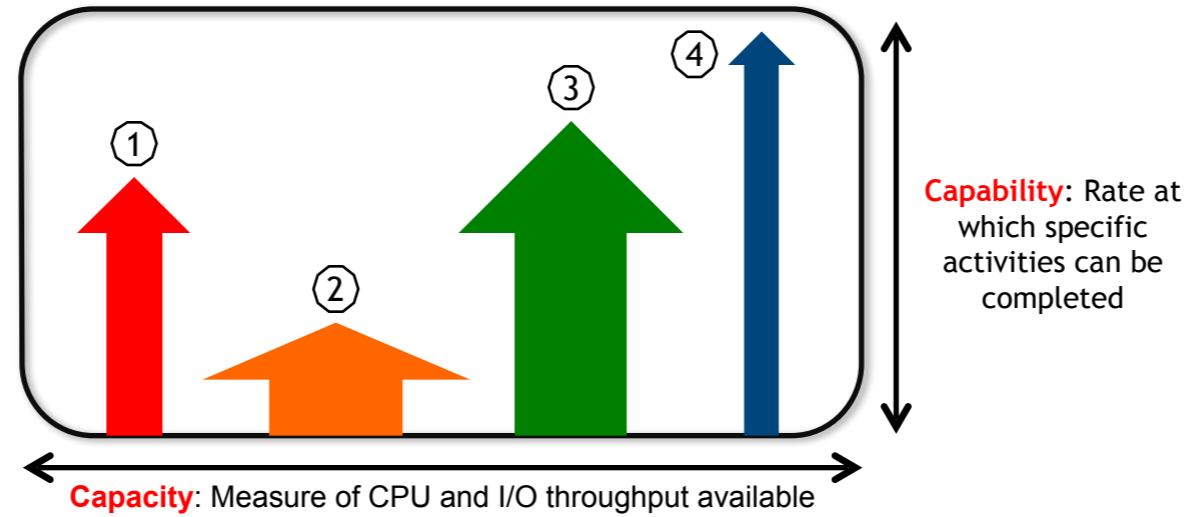
In the picture above, we represent a single Database as a rectangular box hosting 4 different and distinct database workloads, each having workload characteristics represented by both height and width. A distinct workload is characterized as a set of CRUD (Create, Read, Update, Delete) operations that exist as a unit for various reasons - usually because they are part of an application flow that works within defined schema and transaction boundaries. There may be distinctions at the application module layer as well - in other words an application deals with a distinct set of tables that are part of one schema and carries out an understood set of transactions. This application pattern is amenable for segregation by directing the application servers to connect to a specific Oracle service (by using a named service such as "SERVICE=SRV_<XYZ>" handle in the TNS entry). Schema boundaries also provide another means of segregating and isolating workload. For the purpose of describing the above diagram, our assumption is that we can segregate the applications by schema and/or transactions and are using Oracle services for segregation.

Assume the capacity of the database server is represented by the width, i.e. the width of the box is a representation of amount of CPU and I/O work that the server is capable of - in other words, the *capacity* that we indicated earlier. Also, let's assume that the height of the box represents the speed at which various workloads can be processed, i.e. the *capability* of the box. Ultimately, this translates to the *rate* at which the CPU and the various subsystems are able to repeatedly modify a single memory location. Since this memory location should be readable and writable by only one session at a time for consistency's sake, we need to have some mechanism to control concurrent access. This is usually implemented using mutexes, latches and locks - all of them well known concurrency control mechanisms.

Scalability patterns



Different components in a Database Workload – Single node Database server



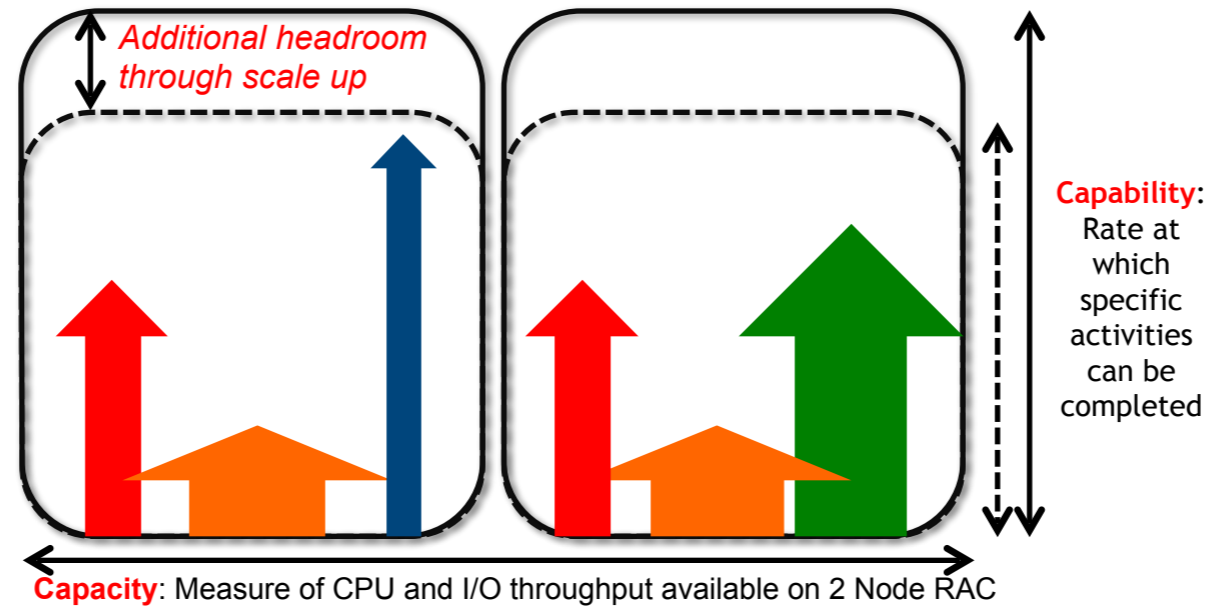
1. Red arrow: Medium capacity, medium rate of work, can be scaled out
2. Orange arrow: Higher capacity, low rate of work, can be scaled out
3. Green arrow: Higher capacity, higher rate of work, scale out questionable
4. Blue arrow: Low capacity, but extreme rate of work, scale out difficult

Confidential and Proprietary

Let us examine this concurrency requirement in greater detail since this is an important point. A good example of concurrent access is that of obtaining and holding a cache buffer chain latch in order to walk a chain of blocks in a hash bucket to get to a specific block on that list. If a large number of sessions need to *concurrently* read or write to that block, then the throughput of that application flow or set of transactions performing that function is constrained by the speed at which sessions can walk that chain when holding the latch. To provide better concurrency, either the number of sessions should be reduced, or you should employ faster CPU's that can access RAM much faster. This scenario is exactly where faster hardware (scaling up) can help, but only to a certain point. As an aside, as the number of processes increases, and need for concurrent access proportionally increases as well, the number of processing spinning on CPU increases and the CPU load increases as a result, since the number of spins (based on the infamous “_spin_count” in the case of latches or the new “_kgx-spin_count” for mutex) occurs without processes yielding the CPU - this is particularly dangerous, and is expressed as disproportionately increasing CPU usage and load as concurrency for a particular object increases. [Note that Oracle has introduced some optimizations in holding a latch in shared mode - this has mitigated the concurrency requirement somewhat]

Scalability patterns

Different components in a Database Workload – Scaling using 2 Node Oracle RAC cluster



- Scale **out** using Oracle RAC (2 Node in this case)
- Scale **up** using larger/more powerful Database Servers



FACTors that affect scalability

- Three simple factors that directly affect scalability:
 - Efficiency of code paths that provide particular functionality
 - Additional paths code needs to traverse to provide concurrency
 - Speed with which code paths can be traversed (both for concurrent/normal cases)
- How do we enhance or increase scalability?
 - Use efficient code paths: Software optimizations and improvements in both Oracle code and Application/SQL/Object design
 - Optimize concurrency: E.g. spin gets and back-off for latches, post-get for locks, “spread the pain” approach [our previous example]
 - Employ faster hardware: Faster CPUs, Faster L1/L2/L3 cache and RAM, SSDs, Hardware based I/O accelerators, etc.



Oracle database specific factors

- Optimizations and feature adds over the years:
 - Introduction of PL/SQL in Oracle 6.0.33
 - Oracle Parallel Server 6.2 (first ancestor of Oracle RAC)
 - Introduction of CBO (specifically Hash joins) in Oracle 7
 - Partitioning and Advanced Queues in Oracle 8.0
 - Substantial improvements in CBO and related ecosystem since Oracle Database 10g
 - Enhanced statistics collections
 - SQL Profiles/Baselines/Adaptive Cursor sharing in 10g/11g
 - Adaptive Optimization in Oracle Database 12c
- Inherent optimization due to efficient codepaths/fixes



Oracle Optimizations you can use

- Partitioning and Index enhancements
 - Use Range + sub-partitioning by hash: Leaner data, efficient indexing
 - Use Global Hash Partitioned index for Insert heavy tables
 - Selective index creation on partitions (new feature in Oracle DB 12c)
 - IOT's to co-locate rows that arrive at different times but are frequently accessed together (typically "saga data")
- SQL and PL/SQL improvements
 - Analytical SQL: New and innovative ways to rewrite aggregation and "windowing" queries
 - Use PL/SQL to bring processing closer to the Database (reduce network round-trips to remote clients; avoid "row by row" fetches)
 - Client side and PL/SQL Result set caching can reduce access contention as well as network round trips




Oracle Optimizations you can use

- **Advanced Queues (Oracle AQ)**
 - Disengage asynchronous portions of the workload (process later locally or remotely)
 - Provide more “scalability runway” by doing only essential (synchronous) work at the “edge of the application”
 - Provide one-to-one, one-to-many, many-to-many data transport efficiently inter- and intra-database
- **Golden Gate Replication**
 - Transport/transform data quickly and efficiently from Live to downstream/reporting/analytics database
 - Cross-platform replication, Upgrades, Asynchronous processing, etc.
- **Active Data Guard**
 - Offload reads from read-write primary



Oracle Optimizations you can use


- Oracle RAC
 - Provides well known scale out method (increases availability as well)
 - Understand the caveats and work around them before deploying
- Oracle Appliances
 - Oracle Exadata: Smart scans, Storage Indexes, Cell based I/O offload
 - Many new features coming up
 - “All in the same box” optimizations
- In-Memory Caching
 - New option in Oracle Database 12c
 - Accommodate OLTP and Batch in the same database (e.g. Oracle EBS)



Scalability anti-patterns

- **Overuse/abuse of concurrent operations: Main anti-pattern**
 - We already noted the “right growing” index example
 - Inefficient SQL plans (typically using tight NL joins causing CPU load)
 - Negative effect multiplied by large session count (causing CBC latching)
 - Not using bind-variables for SQLs (hard parsing, shared pool usage)
 - Concurrent updates to a small set of rows across multiple sessions
 - Problems may end up amplified by Oracle RAC
- **Unnecessary/Incorrect use of indexes**
 - E.g. Creating index on “time truncated” DATE values
 - Creating indexes on columns which already lead in a composite index
 - See <http://richardfoote.wordpress.com/> for many articles on indexing

- **No Information Lifecycle Management**
 - “buzz word” for not purging/archiving older, colder data
 - Range or Interval partitioning key to enabling efficient purging
 - Remember: INSERT scalability is enhanced through hash (and sub-hash) partitioning!
- **Using ORDERED sequences in Oracle RAC environments**
 - Ensure code/application is NOT dependent on ordered IDs
 - Avoid side-effect of monotonically increasing IDs on primary/other indexes
- **Mixing OLTP and Batch in the same database**
 - Oracle EBS is a good example
 - May sometimes be necessary, but look at ADG offloading for reads



Scalability anti-patterns

- Using central, single table based workloads
 - E..g. FND_CONCURRENT_REQUESTS table in Oracle EBS
 - Worsens in RAC – “gc current” and “gc cr” waits
- Increasing the “thread count” at App/Mid tier
 - This is actually a very common pattern, brought about by many factors
 - Usually a knee-jerk reaction, done without DBA’s knowledge/agreement
 - If scalability bottleneck is concurrency, problem becomes worse
 - Solution is usually application and/or SQL/code redesign
 - Employ SESSIONS_PER_USER limit (DBA_PROFILES) to control such “outside database” changes
 - Clients get “ORA-02391: exceeded simultaneous SESSIONS_PER_USER limit” so they need to come to the DBA...



Scalability anti-patterns

- **Not isolating schemas and workloads**
 - Good data modeling and schema, transaction and code isolation is essential
 - Provide logical and physical separation, using mechanisms such as AQ
 - Connect distinct applications, modules using Oracle Services
- **Not designing for reads as well as writes**
 - E.g. Hash (and sub) partitioning for write performance impacts reads
- **Not building instrumentation, monitoring and alerting**
 - Build instrumentation for troubleshooting and logging
- **Testing on a dataset that is either tiny or not representative**
 - Scalability and other issues do not show up in testing



THANK YOU