SYBASE
TECHWAVE
SYMPOSIUM 2009

SQL Anywhere 11 Performance Analysis

Glenn Paulley
Director, Engineering
Sybase iAnywhere
http://iablog.sybase.com/paulley

Agenda

- Overview
- Demonstration of automatic application profiling
- Details about application profiling
  - Architecture
  - User interface
- Demonstration of manual application profiling
  - How to set it up
- Other tools for troubleshooting performance
- Advanced manual diagnostic analysis
- Troubleshooting specific performance problems
- Questions?

Our problem

- Users are complaining about slowness on our application – what should we do?
  - We will use a “sabotaged” version of SalesSim
  - Simulates the sales, shipping, and finance departments of a company
  - How do we use the new profiling features of SQL Anywhere 10 to find the boat anchors and restore/improve performance?

Application profiling

- Combines in one tool most of the functionality provided by:
  - Request logging
  - Procedure profiling
  - Graphical plan capturing
  - Index consultant
  - Statistics monitoring
- Many usage scenarios:
  - Debugging application logic
  - Troubleshooting specific performance problem
  - Leave running in background permanently

Application profiling wizard

- SA Plugin for Sybase Central includes the Application Profiling Wizard
  - Handle all details of setting up a profile and analyzing it
  - Detect common problems automatically
    - Schema
    - Indexes
    - Server and connection options
    - Application structure
  - Make suggestions for improving your application
  - Simplest way to use application profiling capabilities

Manual application profiling

- Using the application profiling tool manually allows for more flexibility in controlling what data is analyzed
- High level steps:
  - Create and start a tracing database
  - Configure a tracing session
  - Run your application
  - Close ("detach") the tracing session and save the trace data
  - Analyze the tracing session using the application profiling mode in the SQL Anywhere plugin

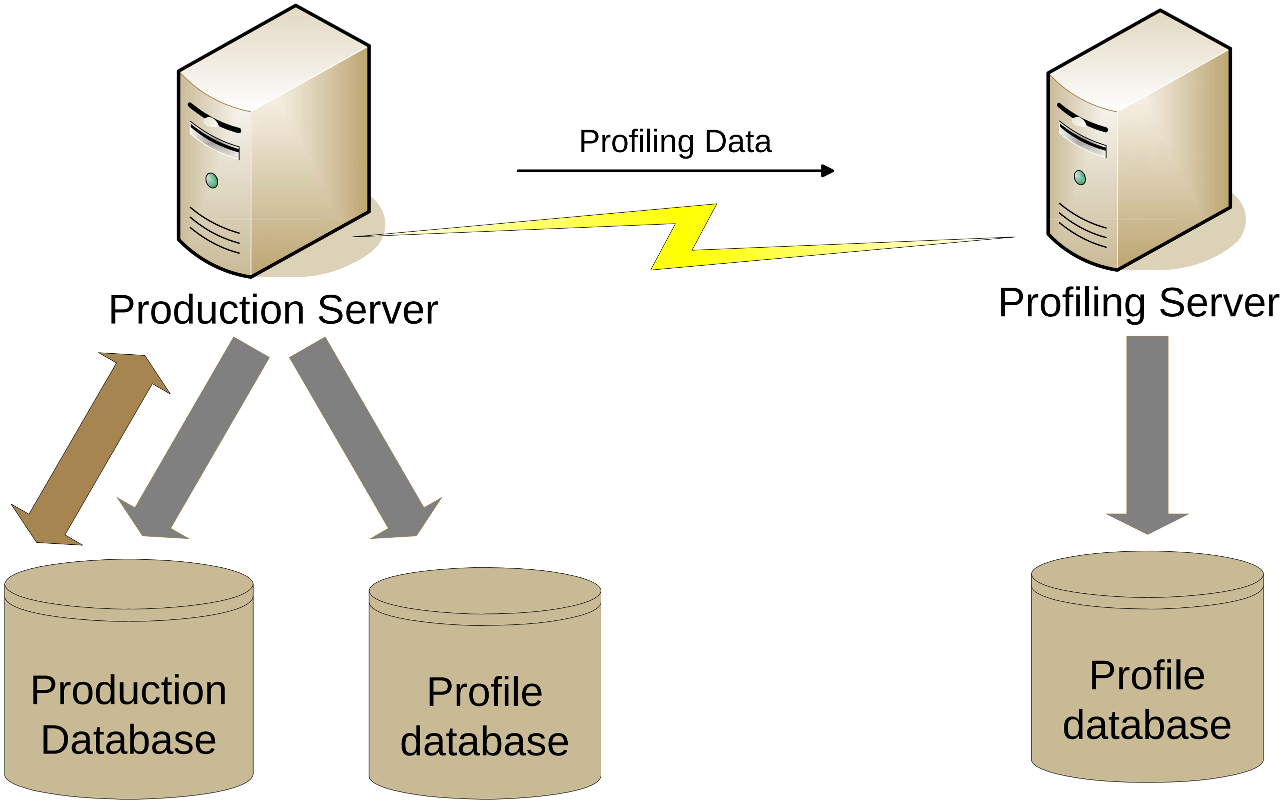
Diagnostic tracing

- Engine in version 10 and up includes functionality to record many types of database events:
  - Connections
  - SQL statements
  - Query execution plans
  - Blocked connections
  - Deadlocks
  - Performance counters
- All types of data can be traced from sources both internal and external to the server

Diagnostic architecture

- Traced data can go to any database
  - To local database for ease of use
  - To a non-local database for performance and to avoid bloat
  - For best results, use a dedicated database
- Traced data stored in temporary tables
  - New feature in version 10: shared temporary tables
  - No I/O overhead
  - At end of logging session, data automatically saved to permanent storage (base tables)

Diagnostic architecture



Specifying what to trace

- Trace only for a specified list of objects:
  - Users
  - Connections
  - Procedures, triggers, functions, events
- Trace only under certain circumstances
  - When a statement is “expensive”
  - When a query differs from its estimated cost
  - Every n milliseconds
- Limit volume of trace that is stored
  - By disk space
  - By length of time

Specifying what to trace

- You can mix and match these configurations and change them on the fly
- For example:
  - Trace all plans used by user ‘ALICE’
  - Trace all statements used by procedures ‘PR1’ and ‘PR2’
  - Trace all query plans in the database for queries that take more than 20 seconds
- You can use the default tracing levels (low, medium, high) as a template
  - The tracing wizard in the SA Plugin will give you this choice
  - Manually, using the sa_set_tracing_level() procedure

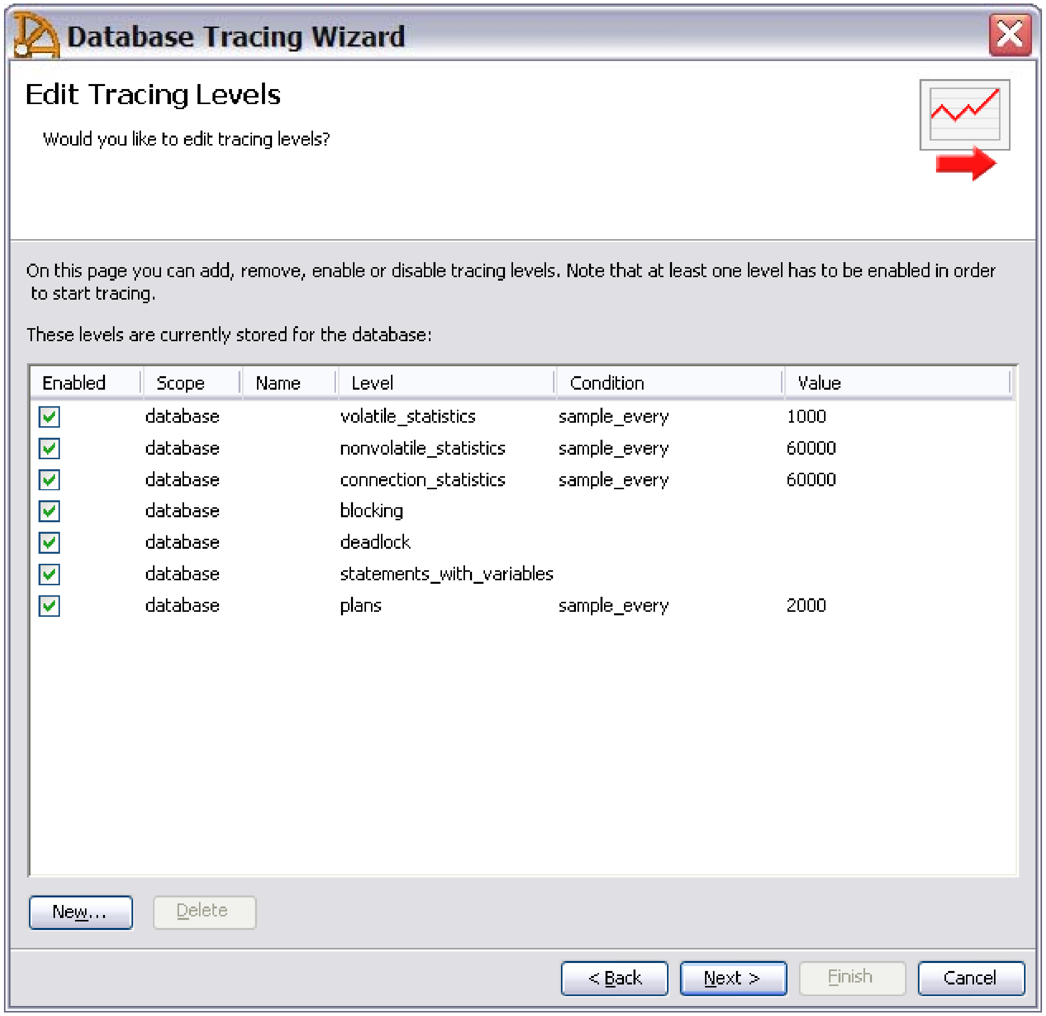
Controlling tracing manually

- sa_set_tracing_level()
- ATTACH TRACING TO ‘connstr’
- [LIMIT {HISTORY nnn{MINUTES|HOURS|DAYS}}
- | {SIZE nnn{MB|GB}}]
- DETACH TRACING {WITH|WITHOUT} SAVE
- REFRESH TRACING LEVELS

The sa_diagnostic_tracing_levels table

- Scope – what objects are we interested in?
  - The whole database?
  - A specific procedure, user, connection, or table?
- Type – what type of data are we interested in?
  - SQL statements?
  - Query plans?
  - Information about blocks, deadlocks, or statistics
- Condition – under what conditions should we capture this data?
  - Only for expensive or misestimated queries?

Specifying what to trace



Edit Tracing Levels

Would you like to edit tracing levels?

On this page you can add, remove, enable or disable tracing levels. Note that at least one level has to be enabled in order to start tracing.

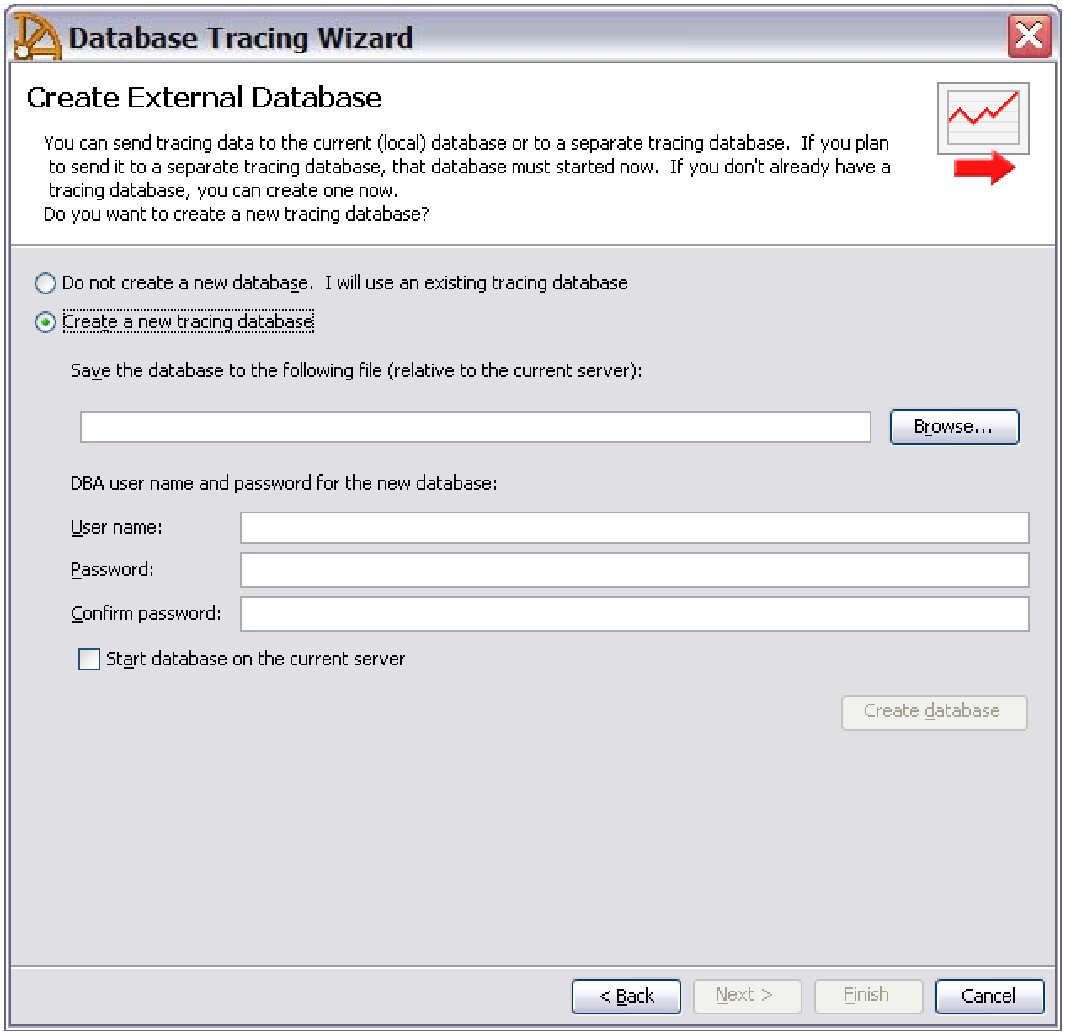
These levels are currently stored for the database:

Enabled	Scope	Name	Level	Condition	Value
☑	database		volatile_statistics	sample_every	1000
☑	database		nonvolatile_statistics	sample_every	60000
☑	database		connection_statistics	sample_every	60000
☑	database		blocking		
☑	database		deadlock		
☑	database		statements_with_variables		
☑	database		plans	sample_every	2000

Controlling tracing with the SA Plugin

- The Tracing wizard is accessible by right-clicking on the database object
  - First, choose basic tracing level – it acts as a template
  - Then, add or remove specific tracing entries
  - Next, if you need to create a tracing database, create it and start it on a database server
  - Finally, specify where the trace is to be sent, and how much data to store

Tracing databases



Create External Database

You can send tracing data to the current (local) database or to a separate tracing database. If you plan to send it to a separate tracing database, that database must started now. If you don't already have a tracing database, you can create one now.
Do you want to create a new tracing database?

○ Do not create a new database. I will use an existing tracing database

● Create a new tracing database

Save the database to the following file (relative to the current server):

DBA user name and password for the new database:

User name:

Password:

Confirm password:

□ Start database on the current server

Saving a tracing session

- When finished tracing, the tracing session can be stopped (detached):
  - In the SA plugin, right-click the database object
  - Manually, use the DETACH TRACING statement
- Detaching without saving will leave the data in the temporary tables in the tracing database
  - It can later be saved using the sa_save_trace_data() procedure
- Detaching with save will permanently store the data

Analysis of traced data

- Can be viewed / queried in real time during trace
  - Using DBISQL or custom scripts, issue queries against the sa_tmp_diagnostic_* tables

- Once saved, a tracing session is analyzed using Application Profiling mode in SA Plugin
  - Provides multiple views of traced data
    - Allows “drill-down” to see more detail about a specific entry
  - Graphical correlation of performance statistics with statements that were active at the time
  - Automatic detection of common performance problems

Replay of server state

- Tracing captures optimizer state as queries are executed
  - Captures cache contents, table sizes, option settings, etc.
  - Allows server to recreate the optimizer state for queries in the trace
    - Not foolproof (because of changing statistics)
  - Can be used to see the graphical plan used by the server when only SQL text was traced
  - Lets Index Consultant make higher-quality recommendations
  - Works even if tracing sent to another database
    - Allows Index Consultant to run offline on another server

Status panel

- If the trace was created as part of the Application Profiling Wizard:
  - Shows a summary of what was captured
  - Performance recommendations are automatically generated and available on the Recommendations panel
- If you created a tracing session manually:
  - Shows all tracing sessions stored in a database
  - Allows you to generate recommendations

Summary panel

- Gives a high-level view of SQL statements captured by the tracing session
- “Similar” statements are grouped together
  - For each statement a signature is computed
  - For queries, insert, update, and delete statements, statements are similar when they involve the same tables and columns
  - Other statements are grouped by type (for example, all CREATE TABLE statements are similar)
- From this view, you can determine which statements are most expensive, either because:
  - They are expensive individually, or
  - They are cheap individually but executed many times

Details panel

- Shows low-level details about all SQL statements captured in the trace
  - Start time is the time the statement began execution
  - Duration is the amount of time spent by the server processing the request – all statements have a minimum reported duration of 1ms
  - For cursors, time the cursor was closed
  - For compound statements, shows line number and procedure name (if available)
  - Text plan is always captured at optimization time

Statement details

- Right click on a statement to obtain more details about it
  - User that executed it
  - SQL error code, if any
  - SQL text
    - If the statement was captured as it was executed, the text will be the original text
    - If the statement was captured later (because it met some condition), it will be reconstructed from the parse tree
    - Reconstructed statements may not be identical to the original statements

Query details

- Right clicking on a query from the Details view will show both statement and query details
- Query details include
  - Numbers of each type of fetch (forward, reverse, absolute)
  - Time to fetch first row
  - Text plan captured at execution time
  - Graphical plan representation
    - May be the graphical plan at execution time
    - May be a best guess at the execution plan, based on the conditions in the server – compare a guessed graphical plan to the text plan before relying on it

Blocking panel

- Shows connections that were blocked
  - What statement was the connection executing when it was blocked
  - What connection blocked it
  - How long did the block last
  - Right click to see more details about either connection involved in a block

Deadlock panel

- Shows deadlock events that were traced
  - Displays a graphical representation of which connections waited on each other
  - Shows which connection was rolled back
  - If available (that is, if tracing was attached to the local database), shows the primary key of each row that was blocked on

Statistics panel

- Shows a graphical representation of performance counters captured
  - Multiple statistics can be viewed, but only for one connection at a time
  - You are often interested in changes in a statistic (a “knee” in a graph) – “Show Statements” button will filter the list of statements in the Details panel to just those that fit on the visible portion of the graph

Index consultant

- Index consultant can be invoked
  - on the entire database
  - on individual queries from the Details panel
- It is run automatically when application recommendations are generated
  - But it generates more details when run manually

Other tools for troubleshooting performance

- New properties for performance monitoring
- Almost all of the old methods of troubleshooting performance are available in SQL Anywhere 11
  - There are specific circumstances in which the legacy methods may be the best approach

ApproximateCPUTime

- Connection-level property – CPU time accumulated by this connection
- Reasonably accurate most of the time – but still an approximation
- Each CPU contributes to the counter – thus if two connections are maxing out two CPUs for one second, each will have an ApproximateCPUTime value of 1.0
- Use to determine what connection may need to be dropped if the server is dragging (but be careful!)
- Best viewed from DBConsole

Request logging

- Stores SQL text of all requests
- Enable in two ways:
  - -zr command line switch (with -zo to redirect output to a file)
  - sa_server_option(‘RequestLogging’, ‘all’)
- Additional switches let you store data in a cyclical series of files to limit the maximum captured data
- Probably deprecated in future releases

Procedure profiling

- View the times and execution counts of stored procedures
- This feature is now part of application profiling mode in the SA Plugin
- Can be used manually from DBISQL
  - sa_server_option('ProcedureProfiling','on')
  - Analyse with sa_procedure_profile_summary() and sa_procedure_profile() procedures
- Useful for rapid tuning of procedures – it is easy to change the procedure definitions on the fly

Database Application Performance

- “My database application is performing poorly”
- “Database application”
  - Database is often though of as a black-box appendage during design
  - But it is deeply integrated into many applications
- “Performing poorly”
  - Compared to what?
  - Another system that does something similar? How similar?
  - Expectations? They may be legitimate, but based on what?

Advanced Manual Analysis

- Application profiling browser in Sybase Central shows several views of data
- You may want to do manual analysis for several reasons
  - Performance; especially for more than a million captured requests
  - Ability to detect complex patterns
  - Automation for regression testing
- Do this by querying data stored in sa_diagnostic_* tables directly
  - Note: these tables in these slides will be referred to in **BOLDFACE** and without a prefix
- Can also query against sa_tmp_diagnostic_* version in the middle of a tracing session

Logging sessions

- Diagnostic profiling allows multiple logging sessions to be saved to the same database
  - Primarily for convenience
- All rows in every diagnostic table identified with a logging_session_id
- All joins must include this value

sa_diagnostic_request

- One row added for each diagnostic “request” – might more accurately be described an “interesting event”
- Request types:
  - 1 – new diagnostic tracing session started
  - 2 – statement execution
  - 3 – cursor open
  - 4 – cursor close
  - 5 – client connected
  - 6 – client disconnected
- start_time is time request began executing
- finish_time is time cursor was opened (type 3) or request was finished (all other types)
- Clock resolution trustworthy above 5ms

sa_diagnostic_connection

- One row for each connection that did anything during tracing session:
  - Connected or disconnected
  - Executed SQL
  - Had performance statistics collected for it
- If a connection does none of the above during a tracing session, it won’t appear in the table for that session

sa_diagnostic_statement

- Contains details about the actual text of SQL statements processed during the tracing session
- Each statement has a signature (hash value)
  - Every SELECT, INSERT, UPDATE, or DELETE statement gets a unique hash
  - All other statements have a hash based on their type
- Compound statements are logged in addition to all of their member statements

sa_diagnostic_query

- A row is added to this table every time the query optimizer is invoked
- Records statistics used by the optimizer during query optimization process
- start_time is the time at which the optimizer began the optimization process
  - For a query that opens a cursor, a row will be added to the query table after the row to the request table, but before the row to the cursor table
- Plans recorded at optimization time are in plan_explain and plan_xml

sa_diagnostic_cursor

- A row is added for every cursor opened
- Row is updated once the cursor closes with additional data:
  - Total number of rows fetched
  - Types of fetches (absolute, relative)
  - Total time spent actively processing query – can be compared to difference between cursor open and close time to find cursors that are held open (unnecessarily) long
  - Graphical plan (plan_xml), if captured, will be plan with statistics

sa_diagnostic_blocking

- A row is added to this table every time a connection is blocked while waiting for a lock
- You can determine the statement that was blocked by joining to the request table
- You can make a guess at the statement that caused the blocking connection to hold its lock by looking for all statements and cursors that started before the block and ended after it
- The table and rowid are recorded so you can look at the row in the original database

sa_diagnostic_deadlock

- Every time a deadlock event happens, multiple rows will be added to this table
  - One row for each connection which formed the deadlock cycle
- Information captured is similar to the **BLOCKING** table
- Deadlock victim can be determined by joining to request table and looking for non-zero SQLCODE return value

sa_diagnostic_statistics

- Rows for each sample
- Server, database, and connection statistics all stored in same table, identified by counter_type:
  - 0 = server
  - 1 = database
  - 2 = connection
- For database file properties, connection_number is dbfile number
- Can find counter name using the property_name() database function

Life-cycle of a statement

- EXECUTE request added to **REQUEST** table; start_time is the time at which internal server execution begins
- SQL text added to **STATEMENT** table if it is a new statement (from a client or a procedure that has never executed)
- If the statement requires invocation of the query optimizer, a row is added to the **QUERY** table
- When the statement completes, finish_time and duration_ms updated in **REQUEST** table

Life-cycle of a cursor

- OPEN Request added to **REQUEST** table
- Statement added to **STATEMENT** table
- If optimizer invoked, row added to **QUERY** table
- Once cursor opened, row added to **CURSOR** table
- OPEN request updated – finish_time and duration_ms show time spent optimizing and opening cursor
- When cursor closed, **CURSOR** row updated with total number of fetches and processing time
- CLOSE request added to **REQUEST** table – total time is time the cursor was opened

Analysis scenarios by example

- How do we analyze this data? Consider three examples:
- Profiling mode recommendations:
  - Detection of client-side join
  - Based on the actual query used by profiling mode in Sybase Central:
- Example ad hoc user queries:
  - Which user causes the most blocks?
  - Which queries are common to the most users?

Client-side join

- Application code repeatedly issues same statement with slight variations
    - Might be from client or even within a stored procedure
    - When doing a join, let the database engine do it!

```
SELECT TOP 40 connection_number, MIN(numexecs), MIN(mintime), MIN(maxtime),
    MIN(os.statement_text) FROM (     SELECT r.connection_number connection_number,
       MIN(r.start_time) mintime,  MAX(r.start_time) maxtime,
       MINUTES(r.start_time) mingroup, signature, COUNT(*) numexecs
       FROM dbo.sa_diagnostic_request r KEY JOIN dbo.sa_diagnostic_cursor c
    KEY JOIN dbo.sa_diagnostic_query q
    JOIN dbo.sa_diagnostic_statement s ON s.logging_session_id = q.logging_session_id AND
    s.statement_id = q.statement_id WHERE s.line_number IS NULL
    AND s.logging_session_id = 1 GROUP BY connection_number, mingroup, signature
    HAVING numexecs > 30 ) dt, dbo.sa_diagnostic_statement os
    WHERE os.signature = dt.signature AND os.logging_session_id = 1
    GROUP BY connection_number, dt.signature
    ORDER BY MIN(dt.numexecs) DESC
```

Which users cause blocks?

- Does one particular user acquire locks on many rows?

-
  ```
  SELECT count(*) bcount, user_name
  FROM sa_diagnostic_blocking b, sa_diagnostic_connection c
  WHERE b.logging_session_id = c.logging_session_id AND
  b.logging_session_id = 1
  GROUP BY user_name
  ORDER BY bcount DESC
  ```

Which queries are common to all users

- SELECT count(distinct user_name) AS num, statement_text
  FROM sa_diagnostic_statement s, sa_diagnostic_request r, sa_diagnostic_connection c
  WHERE s.logging_session_id = r.logging_session_id and s.statement_id = r.statement_id and c.logging_session_id = r.logging_session_id AND r.logging_session_id = 1 AND c.connection_number = r.connection_number
  GROUP BY statement_text

Benchmark development

- A detailed application trace can form basis for benchmarking efforts for a specific application
- Find out what are the most expensive statements used by application
  - Most expensive by time
  - Ones that cause the most contention
- Build set of benchmark statements based around these
- If data grows as more clients are added, this becomes much more difficult

Detecting performance problems

- More systematic approach to our original problem: "My application is slow – what do I do?!"
- Poor performance happens because some resource is maxed out
- Limiting resource at machine-level:
  - I/O bandwidth
  - CPU cycles
- Machine might have more I/O or CPU available in parallel
  - But server might not be able to use it in parallel → concurrency-bound
- See whitepaper: "Diagnosing Application Performance Issues with SQL Anywhere"

I/O-Bound applications

- How to detect?
  - Server is slow but not CPU-bound
  - In Windows Task Manager, see lots of reads and writes by database server process
  - In perfmon, look at %Idle Time counter for Physical Disk objects – if below 1%, server is likely I/O-bound
- Sanity-check – hard drive making lots of noise, lit up
- I/O-bound applications may require addition of extra disk hardware

Cache size too small

- If server can’t keep frequently-used database pages in buffer pool, thrashing occurs
- Can detect using SQL Anywhere counters in perfmon or tracing:
  - CacheReadTable vs. DiskReadTable
  - CacheReadIndex vs. DiskReadIndex
- These counters are absolute values, so look at growth over a fixed period of time
  - Should see CacheReadTable grow more than 10 times faster than DiskReadTable, CacheReadIndex more than 100 times faster
  - If not, indication that cache size may be too small

Missing indexes

- Properly tuned indexes can greatly reduce I/O requirements
  - Read only the rows needed to satisfy a query, rather than all rows
- Best way to investigate is with Index Consultant
  - Available from SC Profiling Mode or DBISQL
  - Can also experiment manually with Index Consultant using CREATE VIRTUAL INDEX statement

Query processing memory

- Server may not have enough memory to process queries using regular methods, causing it to use expensive low-memory strategies
    - Special case of cache that is too small
    - Likely cause – very high -gn value (above 100) and small cache size
        - Make sure you really need such a high value if you use it
- If this is the problem you may see in profiling mode that queries identified as expensive don't use hash operators
    - Or, if they do use them, the operator details indicate reversion to low memory strategies, or multiple passes

Suboptimal file placement

- Placement of different database files (system dbspace, secondary dbspace, temporary dbspace, transaction log) may be suboptimal
- In applications that update or delete large quantities of data:
  - can help to push user tables into secondary dbspace on a different physical disk → leaves checkpoint log with more bandwidth
  - Make sure transaction log is on its own physical disk
  - Avoid RAID-5 for disks for all types of log!

CPU-Bound applications

- Computation and memory access dominating
- Good news – easy to detect!
- Bad news – many possible causes…
- How to detect?
  - CPU above 98% use for all CPUs assigned to server process
  - Task Manager shows server process consuming all available CPUs assigned to it
  - Make sure no other applications are competing for large amounts of CPU

Suboptimal query plans

- Optimizer is choosing one or more access plans that are substantially poorer than the optimal plan
- Most common causes:
    - Outdated optimizer statistics
    - Incorrect setting of OPTIMIZATION_GOAL option
- To analyze:
    - Capture plans with statistics, either manually with DBISQL, or from a tracing session
    - Look for query operators where estimated number of rows or cost vary drastically from actual values

Suboptimal queries

- Optimizer does the best it can, but sometimes is faced with queries that make it hard to do its job
- Basic cause: server is being asked to compute more data than you really need
- Some common types:
  - Asking for extra columns in a result set that you don't use
  - Frequent calling of user-defined functions (in a predicate, for example)
  - Failing to specify READ ONLY access for queries that are not used for update

Suboptimal query patterns

- Variant of suboptimal queries: server is asked to do more work than is really needed
- Classic case: client-side join
  - Already discussed
  - Simple variant: same query repeatedly issued for the same values
- Solution
  - Look for opportunities to cache values that don’t change on the server
  - Look for ways to combine sets of statement

Concurrency-bound applications

- If application does not seem to be either CPU or I/O bound on the server, it is likely concurrency bound
    - Special case of CPU or I/O-boundedness – application can't take advantage of extra resources
    - ActiveReq performance counter: if high (10 to 20 or higher), indicates concurrency bound
    - If it is low, and CPU and I/O are also low, problem is that application isn't giving server enough work to keep busy
- Concurrency problem may be internal to server, or may be in server-side application code
    - Determine using DBCONSOLE or sa_connection_info() → if connections are blocked on others, application-based concurrency problem; otherwise internal server-based

Internal server concurrency

- Server maintains internal locks to protect server structures
- Contention for these resources may happen (less likely for version 11):
  - Transaction log → solution: move to new disk, pregrow
  - Checkpoint log → solution: secondary dbspace
  - Lock table – only possible when servicing hundreds of connections
- May also happen because multiprogramming level is too low → solution: increase –gn value

Application concurrency

- User connections hold locks on rows, preventing other connections from getting their work done
- Most common causes:
  - Hot row – detected by application profiling
  - Long term holding of write locks – detected by looking for cursors that stay open for a long time, long transactions
  - Long term holding of write locks → keep update transactions as short as possible
- Snapshot isolation – can be a viable solution to concurrency problems, but introduces extra CPU and I/O overhead

Questions?