SYBASE
TECHWAVE
SYMPOSIUM 2009

SQL Anywhere Application Development Best Practices

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

Goals of this presentation

- To help you develop applications that are
  - robust,
  - well designed,
  - have good performance, and
  - can scale with your database and number of users
- But we won't have time to discuss
  - User interface design
  - Benchmarking and scalability testing
  - Physical database design
  - And lots of other appdev issues

An invitation to YOU

- If you have an application development tip, whether it be server-related, API related, sync related, or whatever:
  - Email it to me:
    - paulley@sybase.com
  - Or post a tip to the newsgroup
    - sybase.public.sqlanywhere.general

Contents

- General considerations:
  - Schema design tips
  - Application development tips
- Some technical details concerning:
  - Isolation levels
  - Cursor support in SQL Anywhere

When should you think about performance and scalability?

- During the design and planning stages
- Capacity planning
- Improving performance for deployed databases

Earlier is better!

Common areas for performance problems

- Physical database organization
  - Database file characteristics
  - Indexing considerations
- Schema design
- Server characteristics
  - CPU, disk activity
- Network characteristics
  - Insufficient bandwith
  - Latency
- Application design
  - Inefficient client-server communication
  - Query complexity
  - Trigger design
  - Locking
  - Workstation processing (CPU, disk)

Schema Design

Schema design

- Define your tables
  - Normalize your data
  - Entity/Relationship (ER) modeling
- Define appropriate primary keys for all tables
  - Helps in replication environments (reduces amount of data in the forward transaction log file)
- Define appropriate foreign key relationships
  - FK relationships are needed for the query optimizer to generate efficient join strategies
- Define appropriate indexes
  - Don't need to create indexes for PKs or FKs
  - Don't over-do it ! Only define ones that are useful
  - SQL Anywhere permits customization of FK indexes
    - Column order, sortedness
  - Can use the Index Consultant to get indexing recommendations

Schema design: primary keys

- Data administration issues:
  - Ensure your application has complete control over key assignment and usage
  - Usually a very bad idea to update a primary key (especially in a replication environment)
    - Don’t use phone numbers, SSN/SIN numbers, or other external identifiers as primary keys
  - It is exceedingly difficult to “hide” primary keys from users (or your customers)
  - Key formats are very difficult to change after deployment

Schema design: primary key generation

- Common problem: large, composite primary keys that are difficult to search efficiently
  - Both retrieval and update performance can suffer
    - Require multiple predicates to search for a single row
    - Group By, Order By operations require multiple columns at the expense of computation speed
    - Indexes require multiple columns, increase index fanout
  - Consider surrogate primary keys; change existing keys into unique constraints or secondary indexes
- Choose the underlying data type carefully
  - Double or float, because they are imprecise, are not good choices

Schema design: primary key generation

- Integer representation is the most efficient, for both storage and indexing
  - More efficient that DECIMAL
  - Permits the use of autoincrement PK column; hence the server does all the work, eliminating the need for sophisticated key generation within the application
  - ROT: autoincrement scales very well; useful in many situations; highly recommended for synchronization
  - Global autoincrement can generate unique PK values in a replicated system (also in UltraLite!)
- But….

Schema design: autoincrement PKs

- Some disadvantages of autoincrement:
  - Often wish to differentiate keys of different business objects
  - May desire randomized key generation for some applications
  - Alphanumeric values can aid in data consistency during data entry
  - Autoincrement cannot support self-checking identifiers
- Think about these tradeoffs when deciding on identifier data types

Self-checking identifiers

- Add additional letter(s) (or number(s)) to an identifier to serve as ‘check’ digits
  - Example: Canadian social insurance numbers
    - 8 digits plus check digit
    - Federal government publishes the check digit algorithm so that financial services companies can validate SIN numbers as necessary
    - Algorithm prevents the transposition of any two digits from being a valid number
  - Can do this for alphanumeric identifiers as well
- GUIDs are unique, but they are cumbersome and not self-checking

Other variations

- American Express credit card numbers
  - Account number is separate from card number
    - Individuals may have multiple cards, supplementary cards
  - Card number is first ten numbers
    - Extra five digits after the account number is the account suffix
    - Suffix is altered in case of a lost card; base card number remains the same
- Canadian postal codes
  - Different variation: rather than being self-checking, they are difficult to type because of their format (e.g. N2L 6R2)
    - Eliminate transposition errors
    - Dramatically reduces incorrectly-addressed mail for the Canadian post office

Advantages of non-integer key formats

- Can differentiate between key business objects simply by key format
  - Can take advantage of the format when dealing with external parties, particularly over the phone
  - Can use AAA-999 for one type of object, 999-AAA for another, 999-AAA-999 for another, etc.
  - If using letters, refrain from using vowels so as not to form obvious (potentially naughty) words
- Can differentiate between *invalid* keys and *unknown* keys
  - Can make a difference in customer service situations

Schema design: PK generation – manual

- How can we generate these identifiers?
- Manually assign key ranges
  - Cumbersome, but can work in some business environments
  - Often prone to data entry errors
    - consider self-checking or alphanumeric identifiers to reduce data entry problems
      - Example: Canadian postal codes
        - e.g. N2L 6R2
- Do we really want to number customers starting at 000000001?

Schema design: PK generation – key table

- Create a separate “key generation” table, with one row per business object
  - To add a new key:
    - Initiate a new connection
    - Compute the next key using the existing one as its basis
    - Update the table, COMMIT immediately
  - Several disadvantages: requires additional connection, logging, locking, possible contention
- However: avoid designs that serialize transactions
  - Such designs will not scale

Schema design: PK generation – key pools

- Create a separate, permanent table (the pool) of potential identifiers for each business object
- Each transaction DELETEs a key from this table, and uses it in the INSERT of the actual object
  - On ROLLBACK, identifier is released back into the pool
  - Re-populating the pool once (nearly)-exhausted can be done using a trigger, or an event

Physical schema design issues

- For high performance, physical column order may be important
  - ROT: place frequently-accessed attributes at the beginning of a row
  - Control how much space a large value is inlined in a row by using the INLINE and PREFIX specification for a string column definition
- The column order of composite foreign key indexes does not have to match the primary key
- Use PCTFREE to mitigate internal fragmentation

Physical schema design: foreign keys

- Foreign keys are essential to the optimization of complex queries
  - Join selectivity and cardinality estimation is much more accurate when foreign key constraints are present
  - Also enable a variety of query rewrite optimizations
  - Moving to Jasper may warrant some analysis of FK and secondary indexes; may wish to take advantage of new index key flexibility
- But tradeoff using declarative referential integrity
  - Downside is the maintenance cost for indexes that are not utilized in query processing
    - Index sharing can reduce this maintenance overhead by eliminating some physical indexes
  - In rare situations, consider eliminating some RI and check constraints once application is fully tested

Physical schema design: Entity-type hierarchies

- ETH: a business object with multiple subtypes, for example:
  - Insurance clients: policy owners, payors, insureds, beneficiaries
  - Investments: stocks, mutual funds, term deposits, cash, bonds
- Can be a very useful data abstraction
- ETH implementation is perhaps the most difficult of schema design choices
  - There are no right answers, only tradeoffs
  - Fully-normalized solutions may be cumbersome, and may involve multiple joins for each access

Physical schema design: Entity-type hierarchies

- Design alternatives:
  - Single physical table, different applicable attributes to each subtype in each row
    - Subtype identifier stored with each row, usually must be verified with a predicate in each and every query
      - Can use views that already have this condition builtin
    - Any projection of the table will include attributes inapplicable to all subtypes, hence lots of NULLs
    - Declaring referential integrity constraints can be more difficult
      - May have two or more targets for the same FK
      - May need to tradeoff the ability for two or more subtypes to share the same foreign key, which may compromise UPDATE processing

Physical schema design: Entity-type hierarchies

- Design alternatives:
  - Multiple physical tables, one per subtype
    - Queries involving only one subtype can be exact
      - No need for additional predicates, projection operations
    - Key generation is more difficult should uniqueness be required across all subtypes
    - Queries that require multiple subtypes will need UNION operations
    - Joins involving two or more subtypes will require OUTER JOINs
      - Will restrict potential processing strategies

Client-server application performance

- It's all about reducing LATENCY
- Two things to remember:
  - There are no right answers, only tradeoffs
  - All processors wait at the same speed

Sources of latency

- Server-side latencies
- Network latency
  - Time it takes to perform a round trip over the wire; can use DBPING to estimate
- Inefficient client-server interactions
  - Too many round trips from the application
    - Not all round trips are due to application API calls; some are sent/received as part of the underlying wire protocols
  - Too much (or too little data) sent over the wire
  - Repeated requests for the same data
  - Re-PREPARE of similar or identical statements instead of reusing them

Latency within the server

- Whenever processing of a request is interrupted, increased latency can result
  - Examples:
    - Latency inherent to a query's execution plan
      - For example, using user-defined functions (UDFs) or sub-selects in a SELECT list
    - Blocking due to lock contention
      - Controlled through application design and the choice of isolation levels
    - Blocking due to contention for internal concurrency control mechanisms on shared server resources

Execution plan latency

- Any interruption to the flow of tuples through a query processing operator will increase the computation's elapsed time; for example:
  - A nested-loop join constantly interrupts the retrieval of rows from both tables
  - Evaluating a subquery for every row of a scan can be extraordinarily expensive
    - The SQL Anywhere server goes to great lengths to mitigate subquery evaluation through memoization and query rewriting
  - Retrieving data from pages in the extension page arena interrupts retrieval of base row segments

Execution plan latency

- How you write a SQL statement does matter
  - Watch for join conditions involving user-defined functions, expressions, or type conversion
  - User defined functions that have queries in them tie the hands of the optimizer and can be inefficient
  - Consider the use of WINDOW functions, rather than nested correlated subqueries, to avoid slow, iterative subquery evaluation
  - Only FETCH and reference necessary tables and columns

Execution plan latency

- Simplify the query's syntax if at all possible
  - Select list aliases are useful to identify common subexpressions (including subqueries)
    - e.g. Select (X+10)/2 as quotient
  - Eliminate unnecessary predicates, DISTINCT processing, joins, etc.
  - Don't replace LEFT OUTER JOINs with a subselect in the SELECT list
    - Subselects cannot be rewritten by the optimizer
    - LEFT OUTER JOINs can be executed in a variety of ways; subselects impose nested-iteration semantics
  - Using user-defined functions (UDFs) in a query can kill query performance
    - Use them when you need to; but understand the tradeoffs

Window functions

- Permit another opportunity to perform GROUP BY on an intermediate result within the same query specification
  - Permits all sorts of complex queries that would otherwise require multiple queries and/or temporary tables to hold intermediate results
- See the whitepaper on http://ianywhere.com/developer
- Evaluation of a query specification's clauses is
  - FROM → WHERE → GROUP BY → HAVING → WINDOW → DISTINCT → ORDER BY

Using WINDOW functions

- Original correlated SQL query:

```
Select o.id, o.order_date, p.*
From sales_order o, sales_order_items s, product p
Where o.id = s.id and s.prod_id = p.id
and p.quantity < (Select max(s2.quantity)
                  From sales_order_items s2
                  Where s2.prod_id = p.id)
Order by p.id, o.id
```

Using WINDOW functions

- Rewritten query using a WINDOW function:

```
Select order qty.id, o.order_date, p.*
From (Select s.id, s.prod_id,
      Max(s.quantity) Over (Partition by s.prod_id
                Order by s.prod_id) as max_q
      From sales_order_items s) as order_qty,
      product p, sales_order o
Where p.id = prod_id and o.id = order_qty.id
      and p.quantity < max_q
Order by p.id, o.id
```

Isolation levels

- Isolation levels only affect behavior of read requests from other connections/transactions; writes always cause locks
- Isolation levels for read requests:
  - 0 (default) - no locking; a latch ensures that the entire row is consistent when retrieved from the disk page
  - 1,2 - lock rows in the query's result, but with level 1 the lock is held only while the cursor is on that row
  - 3 - lock every row read and every insertion point crossed during query execution
  - Snapshot isolation – writers don't block readers, achieved by maintaining copies of modified rows
- Addition/removal of a foreign key row requires a read lock on the primary row

Snapshot isolation support

- Provides read-consistency in the face of concurrent writes from other transactions (e.g. writers do not block readers)
- Enabled by a global database option, allow_snapshot_isolation
- Three new transaction isolation levels:
  - "snapshot" – cleanest semantics, transaction sees a consistent view of the database as of transaction start (the time the first row was accessed)
  - "statement-snapshot" – requires less resources, however each statement sees a consistent state of the database but at different times
  - "readonly-statement-snapshot" – like statement-snapshot, but only for queries; update statements execute at the isolation level specified by the UPDATABLE_STATEMENT_ISOLATION option (default is 0)

Snapshot isolation support

- Usage is not free
  - Old copies of rows are maintained in a “row version store” (part of the database’s temporary dbspace) for as long as necessary to ensure consistency for any transaction
  - Old copies are cleaned up by the database cleaner process
  - Indexes have a mix of “old” and “current” values
    - Can affect the performance of both sequential and index scans
- Setting the isolation level:
  - set option isolation_level = ‘snapshot’
  - set option isolation_level = ‘statement_snapshot’
  - set option isolation_level = ‘readonly_statement_snapshot’
- Or within an ODBC application, use
  - SA_SQL_TXN_SNAPSHOT
  - SA_SQL_TXN_STATEMENT_SNAPSHOT
  - SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT

Snapshot isolation support

- Update conflicts are still possible; update statements use locks just like all other isolation levels
  - Isolation levels can be mixed (but not recommended)
  - Database property VersionStorePages contains the number of pages in the temp file devoted to copies of old rows
    - BLOB values do not reside in the temp file, but remain in the main database file and are reference counted
  - Some restrictions on DDL when snapshot transactions are in progress (ALTER TABLE, etc.)

Isolation levels: recommendations

- Use the isolation level that offers your application the best trade-off of consistency with concurrency
  - NB. nothing is guaranteed at level 0 (“dirty read”)
- For isolation level 3, ensure the server can exploit indexes to limit the amount of locking performed
  - The server’s optimizer will try VERY hard to avoid sequential scans at isolation level 3
- If you must use multiple isolation levels within a transaction
  - Specify ISOLATION LEVEL on a cursor basis instead of modifying the option setting

Execution plan latency: Cursors

- ESQL cursor types:
  - no scroll, dynamic scroll (default), scroll, sensitive, insensitive
- ODBC cursor types
  - static, dynamic, keyset, mixed, forward-only (default)

Cursor semantics

- Cursor semantics are dependent on:
  - Membership sensitivity
  - Value sensitivity
  - Scrollability (forward only or scrollable)
  - Updatability (read-only or updateable)

Cursors – membership sensitivity

- Membership sensitivity:
  - **Insensitive**: result rows are fixed at open; no changes after result set is computed
  - **Repeatable**: result rows will not change once fetched
  - **Sensitive**: result rows will change with respect to concurrent inserts, deletes, and updates
  - **Asensitive**: result rows may or may not change depending on update activity and chosen plan

Cursors – some definitions

- Value-sensitivity:
  - **Insensitive**: data values will not change once the row has been fetched
  - **Sensitive**: data values will change with respect to concurrent updates
  - **Asensitive**: data values may or may not change depending on update activity and chosen plan

Cursor combinations

		Row Membership		
		Asensitive	Sensitive	Insensitive
Values	A	ODBC forward-only, ESQL dynamic	n/a	n/a
	S	n/a	ODBC dynamic, ESQL sensitive	ODBC keyset, ESQL scroll
	I	n/a	n/a	ODBC static, ESQL insensitive

Cursor type can be altered by server to be more restrictive than what was requested

Network latency and performance

- Latency: time it takes for a packet to be received at a different machine once sent
- Throughput: number of bits (bytes) that can be transferred in a given period of time
- LAN: typically 1ms (perhaps less) latency, at least 1MB/sec throughput
- WAN: 5-500 ms latency, 4-200KB/sec throughput
  - These are ballpark estimates

Reducing network latency

- Increase the database server's packet size
  - Default in Version 11 has increased from 1460 to 7300; even larger sizes can be beneficial for large result sets
  - Can improve the performance of large FETCHes and multi-row fetches, or BLOB operations (both retrieval and insertion)
- Use the CommBufferSize connection parameter
  - Alter the packet size only for connections that would benefit from a larger packet size.

Reducing network latency

- Consider altering the ReceiveBufferSize and SendBufferSize TCP/IP parameters
  - Preallocate the amount of memory used by the TCP/IP protocol stack to receive and send packets over the wire
  - Defaults for these values are machine-dependent (OS, driver, card manufacturer)
  - Settings of 65K thru 258K are useful for experimentation

Improving network throughput

- Communication compression may improve throughput between client and server over a modem or WAN:
  - Enable using Compress=YES in client connection string, or –pc server command line switch
  - Packets are compressed before encryption
  - Compressed data can be less than 10% of original size, but depends completely on data and the application
  - Consider increasing packet size to achieve greater compression and less number of packets
  - Compression requires additional ~46K per connection
  - You must analyze your application's performance and verify results
    - Compression requires additional CPU; on LANs, compression costs may outweigh savings in bandwidth

Mitigating network latency

- Make client-server communication more efficient by reducing the number of requests to the server
  - Utilize wide fetches or wide inserts from your application
  - Make use of PREFETCHing for large result sets
  - Locally cache information in your application, rather than re-SELECTing it from the server
  - Combine a set of statements into a batch, or embed the statements within a stored procedure so that only one CALL statement needs to be sent from the application

Mitigating network latency

- Make client-server communication more efficient by reducing the number of requests to the server
  - PREPARE/DESCRIBE once during initialization (or on first use)
    - New in 10.0.1 – client statement caching – hides DROP/PREPARE sequences for identical SQL statements
    - Requires both 10.0.1 or newer client and server software
  - Bind columns whenever possible
    - use SQLBindCol() instead of SQLGetData()
  - Avoid COMMITing after every statement
    - This is the default behavior for both JDBC and ODBC
    - Every COMMIT is a CHECKPOINT if there is no transaction log

Mitigating network latency: prefetch

- Prefetch is designed to reduce communication in a client-server environment by transferring sets of rows to the client in advance of a FETCH request
- Prefetch is ON by default
  - To disable outright: use the DisableMultiRowFetch connection parameter or set the Prefetch option to OFF
  - Prefetch is turned off on cursors declared with sensitive value semantics
- New in Version 11: adaptive prefetching
  - Number of rows prefetched increases or decreases depending on application behaviour
  - Maximum number of rows that will be prefetched is 1000
  - Also controlled by number of rows the application can FETCH in one elapsed second

Mitigating network latency: prefetch

- Adaptive prefetching is enabled for cursors for which all of the following are true:
  - ODBC and OLE DB: FORWARD-ONLY, READ-ONLY (default) cursor types; ESQL: DYNAMIC SCROLL (default), NO SCROLL and INSENSITIVE cursor types; all ADO.Net cursors
  - only FETCH NEXT operations are done (no absolute, relative or backwards fetching)
  - the application does not change the host variable type between fetches and does not use GET DATA to get column data in chunks (but using _one_ GET DATA to get the value is OK)
- In ESQL, use BLOCK n to limit the number of rows prefetched for each FETCH request
  - If n is 0, prefetch is disabled

Mitigating network latency: prefetch

- Connection parameters: PreFetchRows and PreFetchBuffer
  - can specify a per-connection prefetch row limit and a per-process prefetch buffer size
- Prefetch may decrease performance if:
  - Application requires fewer rows than the prefetched amount
  - Application performs FETCH ABSOLUTE, backwards FETCH, or scrolls randomly through the rowset
  - At isolation levels greater than 1, prefetch may introduce additional lock contention

Mitigating network latency: Wide fetches/inserts

- For relatively large result sets, use wide fetches
  - Each API call obtains several rows; explicitly set by the application
    - Prefetching may or may not also occur
  - Number of rows wide fetched is configurable for each interface, including ODBC and JDBC
  - Beware of differences in the underlying wire protocol that affect the implementation (i.e. JConnect)
- With wide (multi-row) inserts:
  - Supported by ESQL, ODBC, JDBC
  - Consider LOAD TABLE where appropriate
  - COMMIT at regular intervals to reduce lock contention, limit size of rollback log

Improving application efficiency

- Use the cursor type appropriate to the application's requirements to permit the use of lower isolation levels and reduce unnecessary locking
  - Use SQLSetStmtOption() to set cursor attributes
  - SQL_CONCURRENCY to read only
  - SQL_CURSOR_TYPE to dynamic or forward-only
- Use the BLOCKING option (coupled with BLOCKING_TIMEOUT option) to specify whether or not an application blocks on a locking conflict, or receives an error
- Avoid DDL in applications (including TRUNCATE TABLE) to avoid implicit COMMITs or CHECKPOINTs

Improving application efficiency

- Remember to drop statements at termination - de-allocate statements with SQLFreeStmt()
- When a cursor is READ ONLY, declare it as such
  - Some semantic optimizations are disabled for updateable cursors, such as join elimination, which can greatly simplify the original request
  - Enables adaptive PREFETCHing of the result set if also declared FORWARD ONLY for certain interfaces (eg ODBC)

Improving application efficiency

- Watch for nested-loop joins within your application
  - OPEN CURSOR FOO FOR SELECT …
  - FETCH FROM FOO INTO …
    - OPEN CURSOR BAR FOR SELECT …
    - FETCH FROM BAR INTO …
- Alternatively: reconstruct the set of nested queries with a single LEFT OUTER JOIN
  - Precise construction depends on application behaviour

Improving application efficiency

- Use OPEN ... WITH HOLD only where appropriate
  - All locks (except on the current row) are released upon COMMIT; no guarantees about the state of the other rows
  - Semantics are unclear if ROLLBACK was issued
    - Contents of the cursor is undefined upon ROLLBACK
    - Consider setting the option ANSI_CLOSE_CURSORS_ON_ROLLBACK to force the closure of all cursors on a ROLLBACK statement

Improving application efficiency

- Estimating result set size
  - Avoid doing so if at all possible
    - Results will not be consistent in the face of concurrent updates
  - At OPEN, SQLCA (sqlerrd[2]) contains an estimate of the result set size from the optimizer
    - Use SQLRowCount() in ODBC
    - If positive, estimate is accurate at the time the query was executed (i.e. single table scan)
    - If negative, estimate is from the optimizer

Improving application efficiency

- Estimating result set size
  - Use the ROW_COUNTS option to return an accurate result
    - For DYNAMIC cursors, query is executed twice
    - Result may still change due to concurrent updates
  - Consider SCROLL or INSENSITIVE cursors instead
    - Result is computed only once
    - INSENSITIVE: result set size is fixed at OPEN
    - SCROLL: perform a FETCH ABSOLUTE n where n is “large enough” to force materialization of the entire result
    - sqlerrd[2] contains (n – result set size)

Conclusions

- In addition to tuning the server, considerable performance gains can be made by reducing latency within SQL statements, within the application, and over the network