UTOUG Training Days 2015

Data Layer Decisions

ORMs, SQL and PL/SQL

The Presenter

- Bill Coulam (bcoulam@yahoo.com)
- Programming C, C++, Java and PL/SQL since 1995
- Custom Oracle design, development, [re]modeling and tuning since 1997
  - Andersen Consulting (PacBell, US West, AT&T)
  - New Global Telecom - Denver, CO
  - The Structure Group - Houston, TX
  - Church of Jesus Christ of Latter Day Saints - SLC, UT

Our Map

- Application Data
- A little history of application tiers
- Options
- The PL/SQL architectural option

Application Data

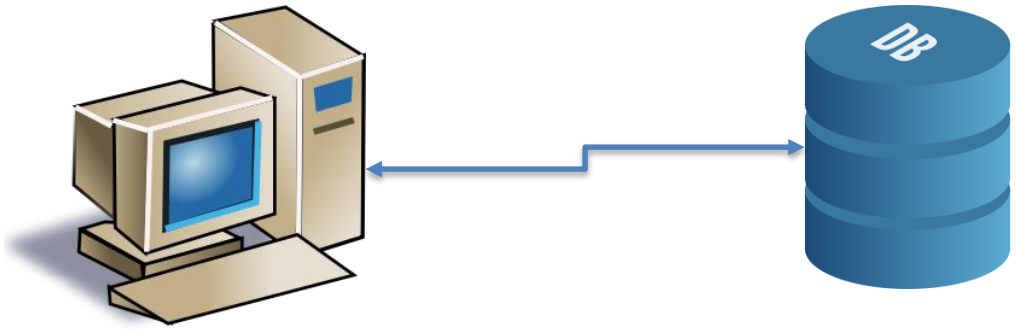
- The only purpose of an application is to capture, validate, store, protect, retrieve, display, share and change data.
- Sticks around forever.
- Almost priceless.
- Must be modeled right, recorded accurately, protected, and reported perfectly.
- Application code falls out of favor and is rewritten and replaced frequently.
- Data, however, is preserved and migrated from release to release.

Data Layer Evolution

- 1-tier (1960s to 1980s)
  - Big iron operating on files
  - Data is application-dependent
  - Reporting required custom development
  - Thin client dumb terminal

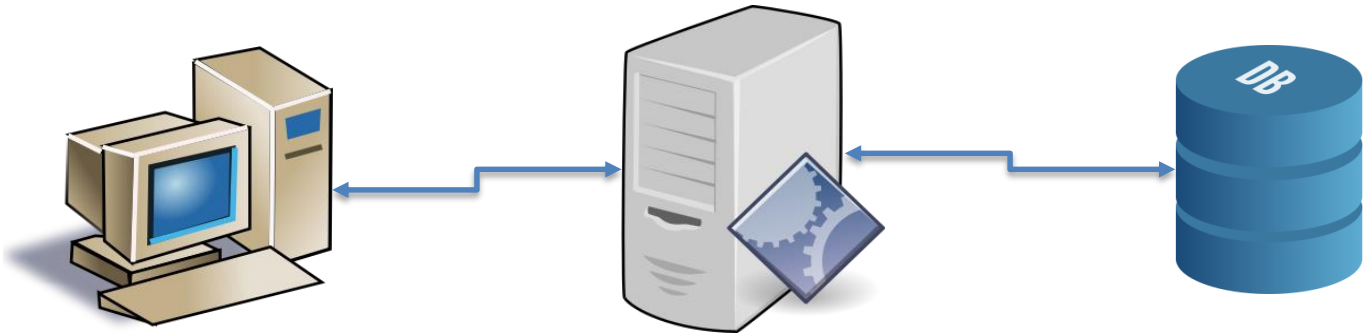
Data Layer Evolution

- 2-tier (80s to 1996-ish)
  - Fat clients held presentation, business and data layers.
  - Large data meant resource intensive clients



Data Layer Evolution

- 3-tier (1997 -) / N-tier (2000 -)
  - Processing farmed out to optimal hardware:
    - Presentation layer in browser and app server.
    - Business rules in app server. MVC.
    - Data rules and queries kept (mostly) in the database



Data Layer Evolution

- 2-tier (2007-ish - now)
  - Devs reject RDBMS, SQL, DBAs, Data Modeling and stored procs
    - Big iron spread out across many boxes, operating on files
    - Data is application dependent
    - Reporting requires custom development
    - Thin client smart browser
- ~~CASE~~, ~~4GL~~, ~~CORBA~~, ~~OODB~~, ~~ERP~~, ~~XML~~, ~~EJB~~, ~~ORM~~, ~~AOP~~, ~~Web Services~~, NoSQL and SOA will save the world

Options for Data Layer

- Custom POJOs and raw JDBC. *Slow dev but screams.*
- ~~EJB: Entity Beans~~ *Ick. Nope. Never again.*
- Big ORM: Hibernate, EclipseLink, TopLink, OpenJPA
- Thin ORM: JOOQ, iBatis/MyBatis, SimpleORM, LinQ, Dapper, more…
- SQLAlchemy, Apache Cayenne, MentaBean, Django, Propel, Entity Framework, .NET Web API, PetaPoco, more…
- Spring JDBC/Template, Spring Data JPA, Spring Sync
- Stored Procedures
- Future: SOA - Service layer on top of queries, views, procs
- Future: Store, retrieve and navigate the objects in the DB

ORMs

- “ORMs are great for simple CRUD operations. As soon as you want to do anything mildly complex or desire efficiency, you need to write SQL.” This is known as a “leaky” abstraction.
- Gavin King says
  - “Just because you're using Hibernate, doesn't mean you have to use it for *everything*. A point I've been making for about ten years now.” - Dec 9, 2013 Gavin King Google+ feed
- ORMs have been called the Vietnam of Computer Science
  - Other articles call them anti-patterns, “the devil” and otherwise show no love
- Devs so fed up with the difficulty of fusing object and data domains, they invent new data access frameworks almost daily
- Can be done right, but usually requires an expert in that ORM

“Thin” ORM

- A “new” flavor of mapping, but map to queries, views and procs. Some map to tables.
- Queries often kept in XML. Easy to maintain and deploy.
- SQL-friendly and DB-centric
- Shallow learning curve
- “Typesafe SQL”
- Supports existing schema, views, procs, etc.

Spring Options

- Spring JDBC
  - LDS Screening Project. On time, under budget, easy to understand and maintain. Fast. No issues.
  - LDS IMOS Project. Certain screens refactored with Spring JDBC saw 100X and 50X speed and resource improvements
- Spring Data JPA
  - Our enterprises' current recommendation
- Spring Sync
  - New. 1.0.0. Utilizes HTTP PATCH for partial updates

SOA

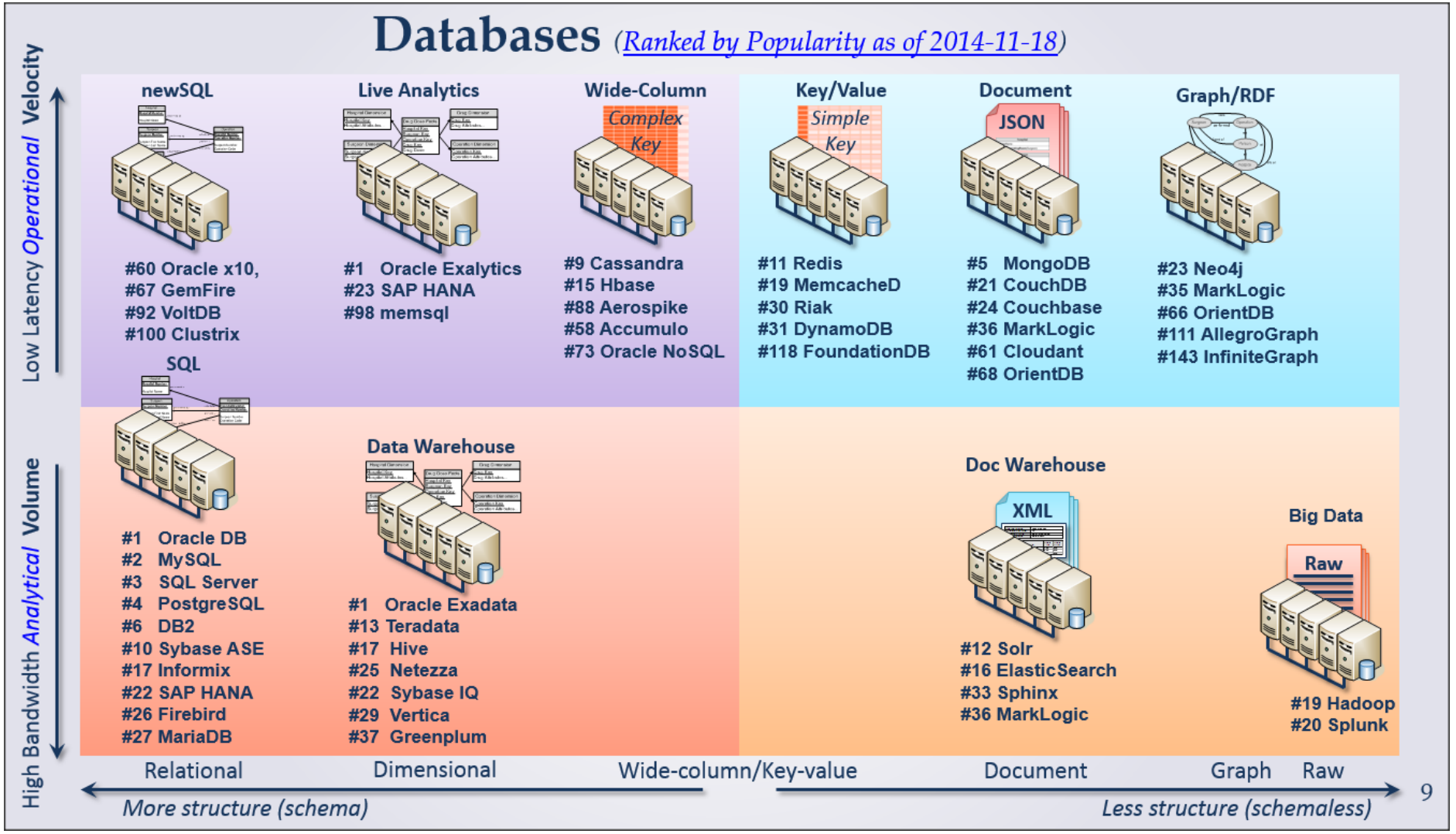
- REST!
- Angular!
- Stateless!
- Mobile web!
- Service Bus!
- New direction being aggressively pursued by many
- Oracle REST Data Services (ORDS)

Ditch Relational

- Store the object graph directly in the database
  - MongoDB, Couchbase
  - MarkLogic (native JSON as of v8)
  - Neo4J and Spring Data Neo4J
  - IBM DB2, PostgreSQL 9.2
  - Azure DocumentDB, Amazon DynamoDB

  And…

  Oracle 12.1.0.2 !



From “Are you ready for NoSQL” by Mike Bowers, Enterprise Database Architect for ICS, Church of Jesus Christ of Latter Day Saints

Stored Procedures

Discuss

Args against PL/SQL

- Inability to port apps to another DB
- Can’t version the code
- In-memory app server logic always faster
- Increased cost, complexity and less maintainability
- No business logic outside of the app server
- DBA bottleneck
- No skillset among the developers. Second lang for app.
- Hard to debug. Rarely tested.
- Not cached. Not pre-compiled. Not faster. Dynamic SQL as fast or faster than stored procs.

Args for PL/SQL

- SQL where it belongs
- Agility when changing app development language/framework
- Continuous and easy deployment
- Speed and efficiency
- Set/Bulk transactions and processing
- ETL
- Security
- Abstraction and de-coupling
- Centralization of common logic and reusability
- Automation of routine data-centric tasks
- Built-in and bolt-on auditing and instrumentation

Where should the SQL rest?

- Religious debate, but I feel there is only truth:
  - Simple SQL is fine in the middle tier
  - Anything harder than a couple joins, store it in a view or packaged PL/SQL routine
- Should be written by someone that understands relational, sets, SQL and the DB
- Reviewed and tuned by DBAs
- Can be easily instrumented for logging and debugging. Much easier to troubleshoot and monitor when kept inside the database, next to the data, where it belongs.

App Evolution

- How often do apps change a framework they're using? (POJO -> EJB -> Struts -> Ajax -> JSF -> Angular -> Angular2 -> ?)
- How often do apps change the database?
- Keeping the SQL and data logic in the database means much less work when re-tooling an app

Security

- If business requires stiff protection of data structures, don’t grant them to any accounts.
- Construct PL/SQL API for data access and manipulation
- Can then design complex security privileges for the stored procs that have access to the tables and views.

Deployment

- Being interpreted, PL/SQL is very simple to deploy.
- Compile the versioned source file into the schema. Done.
  - App data layer should be amended to capture the "existing state of packages has been discarded" error and re-try the previous call
  - Voila. Continuous deployment to Prod achieved.

Data Proximity and Network Latency

- Typical: Query DB, transport result set across the network, load objects, perform the operation(s) on the data in memory, unload objects, transport data back to DB, update and commit.
- Stored Routines: Call routine, query DB, perform operation(s) on the data in memory, update and commit.
- Eliminating the items in red saves significant runtime when dealing with large result sets.

Large Data Sets

- Relational databases, in particular Oracle, are raging speed-demons for querying and manipulating large, normalized result sets.
- RDBMS are set-oriented
- Don’t treat DB like “dumb” persistence box
- Take advantage of the set and performance-oriented features of your DB
  - Doing as much as you can as one SQL statement
  - If PL/SQL is the solution, use bulk features

Extraction

- If requirements need to extract large amounts of data for consumption by app, reporting, data warehouse, external interfaces, etc.
  - Especially if the filter or SELECT features are complex, use PL/SQL

Transformation & Loading

- If requirements need to pump large amounts of data into the database, in particular if calculations or derivations need to read the destination database while processing incoming data, do it in the database using external tables and pipelined PL/SQL functions.

SQL Injection-Proofing

- SQL written by devs and kept in the middle tier is much more likely to suffer from concatenation problems, hurting performance, shared pool and inviting SQL injectors
- SQL behind a PL/SQL interface guarantees no SQL injection (unless dynamic SQL is being written with parameters)

Common Business Logic

- Critical algorithms that must be available to and used by multiple systems and tiers should be kept in the least common denominator (the database) as a stored routine.
- If direct access to the DB is not available, the stored routine can be published as a web service just like any java-based web service.

Reuse

- Database processes, jobs, routines, triggers that need to use common business logic can't take advantage of web services or middle tier methods, which demands duplication and fragility.
- If kept in the database, designed and documented well, and published, encourages re-use, centralization and robustness.

Auditing and Logging

- SQL in the middle tier is notoriously done poorly (by devs or JPA engine).
- When things go wrong, it is difficult to debug, monitor and log.
  - Can be done right, but usually isn't
- SQL kept in stored packages trivial to instrument, monitor, debug, tune and re-deploy.

Automation

- Most common use of PL/SQL
- Used often by physical/system DBAs to schedule routine reports, extractions, loads, monitoring and cleanup tasks, etc.

Considerations

- Time
- SQL expertise
- PL/SQL expertise
- Size and complexity of data model
- Portability
- Scalability and Performance
- Maintainability/Tuning/Troubleshooting
- Persistence engine

Code Examples

- Spring JDBC
- Spring Data JPA
- JOOQ
- Stored Procs for all

bcoulam@yahoo.com

www.dbartisans.com