Query-Directed Data Mining

Techniques using Python & Parallel Processing

Christopher Gillett

Chief Software Architect

Compete, Inc.

Background

- Compete, Inc. analyzes large amounts of data (gigabytes per day), and accrues terabytes of data every year supporting its predictive analysis business
  - Tera-scale storage requirements
  - Massive data processing needs
  - Problem: Ad-hoc research against these large data sources
- Technology platform is Unix clump/cluster/grid of 60+ machines
  - Job level parallelism managed using Portable Batch System
  - Storage is NFS with dedicated NFS servers
  - Storage managed by Compete File System (CFS – presented earlier at PyCon 2005).
  - Application code written in a variety of languages: C, C++, Java, Python, etc.

Characterizing our Data

- **Characterizing Size: Massive data sets, but well-ordered**
  - Massive amount of archived data
  - Terabytes move through our systems monthly
  - Running 1.5 – 2.0 million jobs per year over this data
- **Characterizing Order: Regular Construction**
  - Clearly defined field definitions
  - Easily understood data components:
    - Dates, User IDs, URLs, etc.
- **Characterizing Data Use: Multiple purposes not always in harmony**
  - Conventional data processing:
    - "Collect-Process-Aggregate-Present" cycle done periodically
      - Reduces large amounts of data into smaller more manageable units
      - Resultant work product useful but "less dynamic" than larger raw data
    - Ad-hoc searching and retrieval of data using all data sources
      - Requires efficient processing of potential large amounts of data

Query-Directed Data Mining Overview

- View everything as database – even things that are not in databases
  - Information retrieval and querying done using a query language
  - SQL – or something close to it to manage paradigms not “naturally” in SQL
  - Build language and/or runtime extensions to SQL
    - Provide built-in functions to handle situations unique to our data
    - Extensibility incorporated into system from initial design to full realization

- Benefits of this approach
  - SQL or SQL-like languages well understood and familiar to even casual or novice developers or data professionals
  - These languages have extremely clear semantics which lend themselves well to decomposition into machine-generated code
  - Encapsulated data access and using SQL to retrieve data means tools don’t need to understand any underlying data formats – everything can speak and interact in terms of Schema,

Why Not Just Deploy Oracle and Declare Victory?

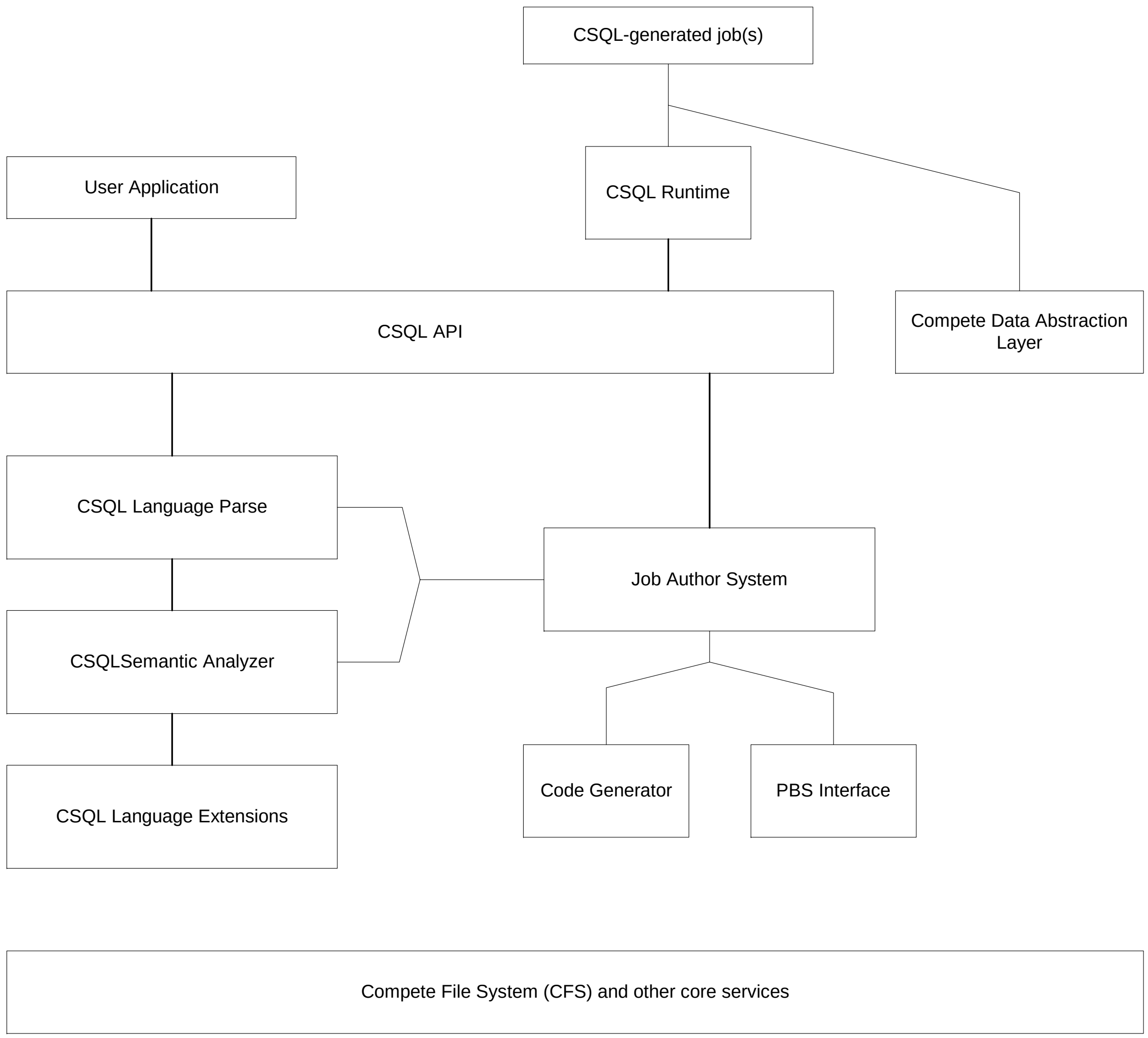
- Full database rollout simply not feasible for our company
  - Seriously expensive proposition for a Stage Zero or Stage One company
  - Cost to deploy Oracle or IBM DB2 can cost hundred of thousands to millions
  - Marketplace peers and competitors have tried this database-centric approach and failed
- We use databases where they make sense
  - Output from our usual collect-aggregate-present cycle is loaded in MySQL databases
  - These databases are used to drive certain production systems
- Conventional databases can "hide behind" our Query-Directed model
  - A query written in SQL obviously works on a database (duh)
  - A front-end system can direct queries to a running database when available

Architecting and Building a Query-Directed System

- Major Components:
  - SQL Language Processor (CSQL – Compete SQL)
    - Provide basic language parsing support
    - Real goal to synthesize intermediate representation (IR) for queries
    - V1.0 essentially implements SELECT
  - Code Generator
    - Abstraction class to be extended by every code generator
    - Allows code generation in language of choice
  - Query decomposition & Job Authoring System
    - Works in conjunction with CSQL processor and code generator
      - Understands logical ways to decompose queries
        - » Across common dimensions – initially date based
      - Understands how to interact with the batch processing system
    - Net effect to allow exploitation of parallel machines at the job level

COMPETE, INC.
The Predictive Edge

CSQL System Overview



Role of Python in the Architecture

- Initially planned to develop 100% in Python
  - Parsing, job generation, and interaction with a Python-based Query Processing Engine
- Python performance limitations with respect to I/O processing made this approach unfeasible
- Revised plan uses Python for “most” of the system
  - Parsing
  - Job Synthesis & Issuing jobs into Portable Batch System
  - Code Generation
  - Limited Python-based runtime for handling certain extensions and built-in functions
    - Some extensions expressed as parser and CG extensions and handled at Query Analysis time
    - Some extensions expressed as calls to a runtime system
      - Acceptable for small data sets
      - Can be painful for large data sets

Leveraging Parallel Resources

- This is the part that's fun
- Common elements in our data hierarchy include date & other distinct discriminators
  - By generating identical queries divided across these common data fields, many jobs can be generated:

    ```
    select * from myTable where date >= 2005-01-01 and date <= 2005-01-31
    ```

    can be rewritten:

    ```
    select * from myTable where date = 2005-01-01
    select * from myTable where date = 2005-01-02 …
    ```

  - In the above example, 31 jobs get generated and can run on 31 machines simultaneously
    - PBS handles the job control
    - CSQL job execution modules handle synchronization and merging of results
    - All this is managed through a fairly simple set of programmer APIs

Interacting with the Query-Directed System

```
### Example 1:  Programmatically running a query
def getData(self):
    query = “select * from myTable where date >= 2005-01-01 and date <= 2005-01-31”
    result = jc.buildFromSQL( query, “myTest”)
    launchResult = jc.launchJobs( result[ "jobname" ] )
    waitForRunningJobs( “myTest” )
```

Extending the CSQL runtime

```
import md5

from compete.csql.sqlFunction import sqlFunction

class sqlBuiltin_MD5(sqlFunction):

    def __init__ (self):
        sqlFunction.__init__( self, "md5", ["s"], "s" )


    def execute (self, args):

        m = md5.md5()

        plainText = args[ 0 ]
        m.update( plainText )
        result = m.hexdigest()
        return result
```

COMPETE, INC.
The Predictive Edge

Implementation, Results and Future Directions

- CSQL System written 100% in Python

- Job Author system built 100% in Python and works in conjunction with CSQL

- Combination used on a variety of mission-critical applications with the company
  - Instrumental in the Compete Data Analysis Workbench
    - Queries used to pull data for common metrics used by data analysts
    - Pleasant aside: Workbench tool 100% wxPython + Python

- Able to comb through terabytes of data representing 3+ years of observations in a reasonable amount of time:
  - Definition of reasonable may vary – this is not a “real time” system
  - Very few degenerate cases
  - Cost of system, even including hardware, a fraction of deploying a commercial database

Future Directions and plans for CSQL

- Current query time performance ranges from good to awful
- Substantial room for performance improvements by improving underlying data representation
  - Such a major overhaul can be done without disturbing higher level applications
- Interest in exploring role for MySQL as a front end to a custom data store

COMPETE, INC.
The Predictive Edge