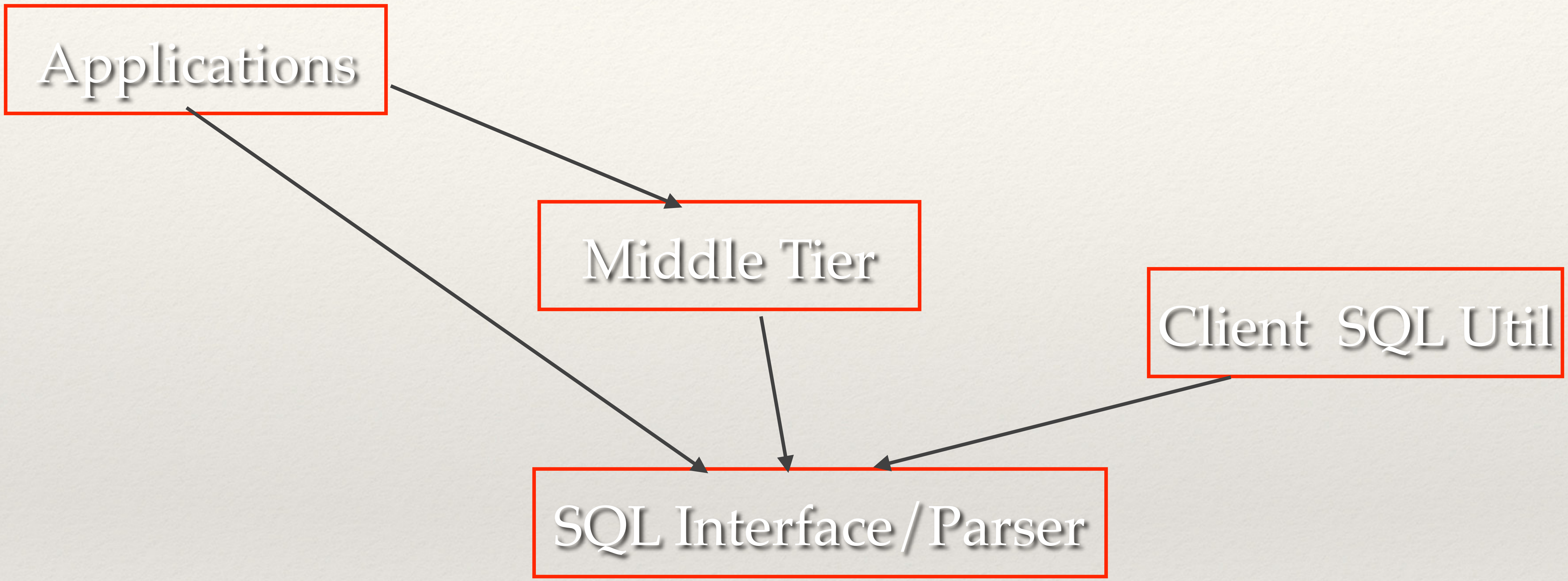
SQL in Programs

Basic Architecture

- Application Layer - what most users see, talks SQL
- Parsing/Planning Layers - the intelligence
- Runtime or execution Layer - the brawn
- Storage Layer - where data resides, may include simple access layer

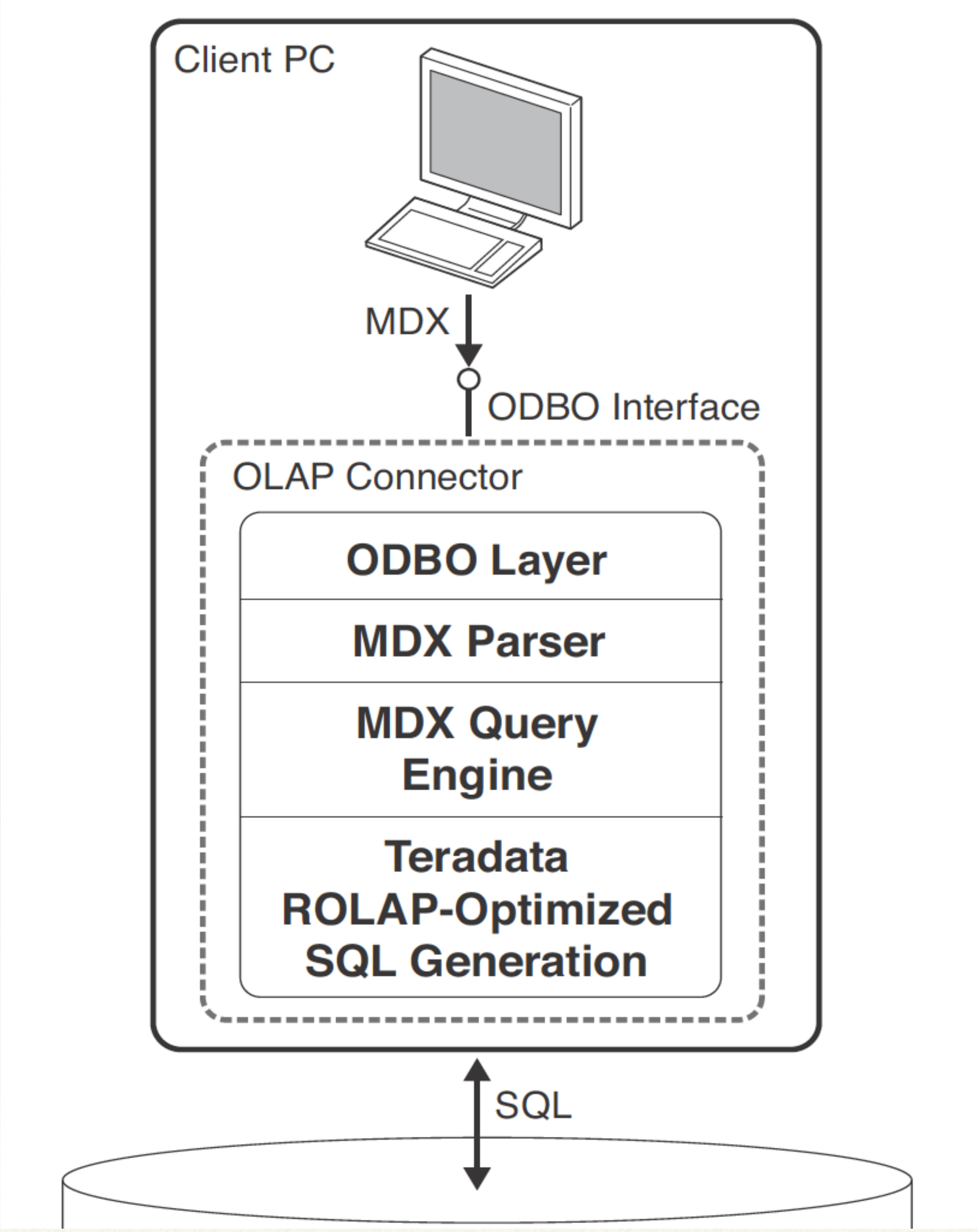
Typical Connections to SQL



- Applications connect via Embedded SQL, ODBC, JDBC, SQLJ, CLI
- Middle Tier: e.g. MicroStrategy, BusinessObjects, etc.
- Client SQL Utilities: psql, Teradata® Bteq, Teradata® SQL Assistant

Connecting Excel to Teradata®

- Excel "talks" MDX/ODBO
- The Connector takes MDX and converts it into SQL (in the client)
- It connects to Teradata® using ODBC
- SQL gets executed by Teradata®



Basic Framework

- SQL calls accessible from a host language, e.g. C or Java
- From the host language one can:
  - Connect to data-source
  - Execute SQL
  - Fetch results
  - Process results

Different API's/Standards

- Embedded SQL standard
- CLI - or Call Level Interface - basis for ODBC
    - Teradata® Bteq (similar to Postgres psql)
- ODBC - Open Database Connectivity
    - Teradata® SQL Assistant, MS Sql Server Mgmt Studio
- JDBC - Java (Open) Database Connectivity
- SQL/J - Sort of Combining Java with Embedded SQL

Overview

- May need to describe the environment the system is running in, e.g. ODBC version
- Connect (login) to the DBMS, and then
  - Execute the SQL's we want (usually need to build it first)
  - Fetch the results
  - Post-process and repeat
- Disconnect from the DBMS

Embedded SQL

- SQL calls are embedded in a host-language, e.g. C
- It's standardized, so same source code will (mostly) work with typical DBMS
- SQL interface precedes with EXEC SQL
- A preprocessor converts the EXEC SQL to calls to the database system
- The final program is compiled as executed with integrated SQL processing

Environment & Connecting

```
EXEC SQL CONNECT TO db1@localhost;
```

- The source code is "preprocessed" by a DBMS specific preprocessor, e.g. `ecpg` for postgres.
- depending on the complexity of connection (local, remote) and security setup, more information may be needed by the `CONNECT TO`.

Shared Variables

```
EXEC SQL BEGIN DECLARE SECTION;
  int sid;
  varchar name[21];
EXEC SQL END DECLARE SECTION;
```

- To communicate, the host language and the SQL share variables
- The data types much match between the two. This is usually well documented and is fairly intuitive, but can vary across systems

Accessing Results via Cursor

```
EXEC SQL DECLARE foo CURSOR FOR SELECT sid,last_name FROM s2;
EXEC SQL OPEN foo;
for (i = 0; i < 10; i++) {
  EXEC SQL FETCH NEXT FROM foo INTO :sid, :name;
  printf("%d\t%s\n", sid, name.arr);
}
EXEC SQL CLOSE foo;
```

- Cursor gives a row-by-row access to the query results
- Then one can fill in the host variables with the row values
- Finally, one should close the cursor
- Cursors can be “updatable” when accessing base tables or updatable views

Put It Together

```
#include <stdio.h>
int main()
{
  int i;
  EXEC SQL BEGIN DECLARE SECTION;
    int sid;
    varchar name[21];
  EXEC SQL END DECLARE SECTION;
  EXEC SQL CONNECT TO db1@localhost;
  EXEC SQL DECLARE foo CURSOR FOR
           SELECT sid,last_name
           FROM s2 ORDER BY sid;
  EXEC SQL OPEN foo;
  for (i = 0; i < 10; i++) {
    EXEC SQL FETCH NEXT FROM foo
                         INTO :sid, :name;
    printf("%d\t%s\n", sid, name.arr);
  }
  EXEC SQL CLOSE foo;
  EXEC SQL DISCONNECT all;
}
```

```
$ ecpg pg1.pgc
$ gcc -o esql pg1.c -lecpg
$ esql
1      Washington
16     Lincoln
32     Roosevelt
33     Truman
34     Dwight
35     Kennedy
36     Johnson
37     Nixon
38     Ford
39     Carter
```

ODBC/JDBC

- A “driver” model introduces another layer of indirection
- The “client” program talks to the driver that then connects to the data-source
- Client program can be fairly independent of the data-source
- Any data-source having an ODBC/JDBC driver interface can be used (and needn’t be an RDBMS)

JDBC

- Java Database Connectivity
- Like ODBC but for Java
- Drivers are available for most databases, e.g. Postgres
    - Provided by DBMS or 3rd party
- JDBC-to-ODBC bridge makes ODBC data-sources accessible

Environment and Connection

```
Class.forName(“org.postgresql.Driver”); //environment
Connection c = DriverManager.getConnection
                             ("jdbc:postgresql://localhost:5432/db1",
                              "ambuj", "123");
```

- First thing is to load the needed Driver
- Then get the connection to “db1” database on postgres
- Logic “ambuj”, password is irrelevant here (depends on security)
- General case below

```
Connection c =
DriverManager.getConnection(“jdbc:somejdbcvendor:otherdataneededbyvendor",
            "myLogin", "myPassword");
```

Create A SQL Statement

```
stmt = c.createStatement();
String sql = "CREATE TABLE MyNewTbl " +
             "(ID INT PRIMARY KEY     NOT NULL," +
             " NAME           TEXT    NOT NULL, " +
             " AGE            INT     NOT NULL, " +
             " ADDRESS        CHAR(50), " +
             " SALARY         REAL)";
stmt.executeUpdate(sql);
stmt.close();
```

- We must create a statement
- And create a SQL that would be part of the statement
  - CREATE TABLE in our example
- executeUpdate - for DDL as no data returned

Insert A Row

```
c.setAutoCommit(false);
stmt = c.createStatement();
String sql = "INSERT INTO MyNewTbl(ID,NAME,AGE,ADDRESS,SALARY) "
             + "VALUES (1, 'Paul', 32, 'California', 20000.00 );";
stmt.executeUpdate(sql);
sql = "INSERT INTO MyNewTbl(ID,NAME,AGE,ADDRESS,SALARY) "
      + "VALUES (2, 'Allen', 25, 'Texas', 15000.00 );";
stmt.executeUpdate(sql);
…
stmt.close();
c.commit();
```

- To Insert a Row, same process but difference SQL
- We can insert many individual rows, using different SQL but same statement
- Commit after all the inserts (otherwise it does after each one)

What about SELECT

```
stmt = c.createStatement();
ResultSet rs = stmt.executeQuery( "SELECT * FROM MyNewTbl;" );
while ( rs.next() ) {
   int id = rs.getInt("id");
   String  name = rs.getString("name");
   int age  = rs.getInt("age");
   String  address = rs.getString("address");
   float salary = rs.getFloat("salary");
   System.out.println( "ID = " + id );
   System.out.println( "NAME = " + name );
   System.out.println( "AGE = " + age );
   System.out.println( "ADDRESS = " + address );
   System.out.println( "SALARY = " + salary );
   System.out.println();
}
rs.close();
stmt.close();
```

- What if we don't know all the data types?

```
$ java PGSelect
Opened database successfully
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

Metadata About A Query

```
ResultSet rs = stmt.executeQuery( "SELECT * FROM MyNewTbl;" );
ResultSetMetaData rsmd = rs.getMetaData();
for (int i = 1; i < rsmd.getColumnCount(); i++){
   System.out.println(rsmd.getColumnTypeName(i)+"\t\t"
                     +rsmd.getColumnName(i));
}
```

- Can get the output Column Types and Names

```
$ java PGMeta
Opened database successfully
int4   id
text   name
int4   age
bpchar   address
float4   salary
Operation done successfully
```

Motivations for SQL Programming

- Interacting with applications that work with data
- Presenting a different view of the data in the DBMS
  - Many BI apps fall in this category
- Doing things that are hard to do in SQL
  - Very common in old days but less so

Issues

- Performance - data fetched serially
- Scale - result dataset too large
- If we want to further use the DBMS, putting data back is equally slow

Solutions

- Move computation to the data
- New DBMS features
  - Computations built into the system
  - Top N eliminates need for ORDER BY and fetch
- Temp Tables, compute and store results in the DBMS
  - Multi-step processing can be done in the DBMS
- Extensibility features, move computations to the DBMS
  - scoring, statistical analysis