Architecture and Interfaces

Shan Hung Wu & DataLab
CS, NTHU

RDBMS

- Definition: A ***Relational DBMS*** (***RDBMS***) is a DBMS that supports the relational model

Outline

- Architecture of an RDBMS
- Query interfaces
  - SQL, JDBC, and native interface
- Storage interface
  - RecordFile and metadata

Outline

- Architecture of an RDBMS
- Query interfaces
  - SQL, JDBC, and native interface
- Storage interface
  - RecordFile and metadata

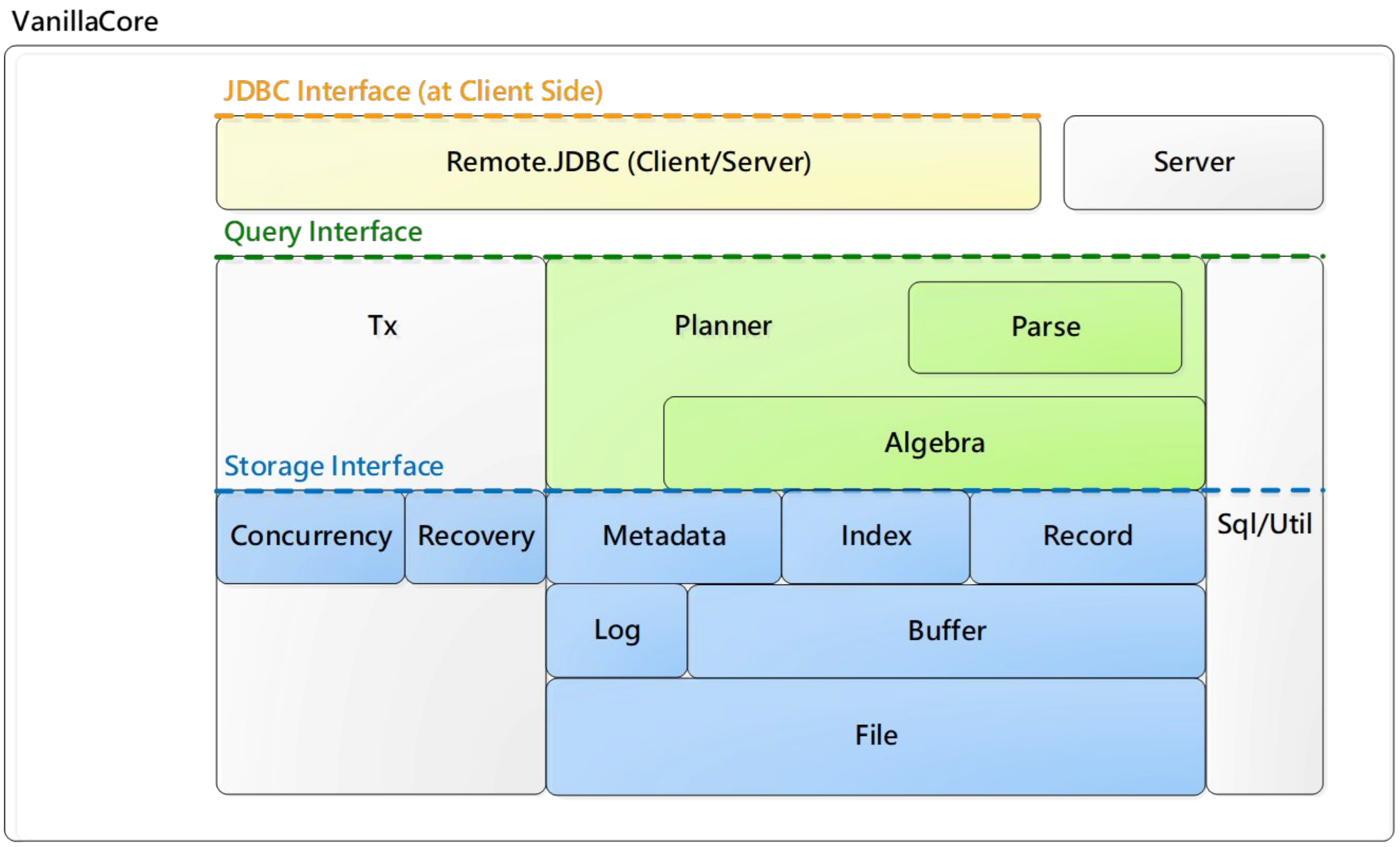
Architecture of an RDBMS

- Largely influenced by the IBM System R
  - Announced in 1974

The VanillaDB Project

- VanillaCore
  - An RDBMS that runs on a single server
  - Written in Java
- VanillaBench
  - Common benchmarks for RMDBs
- VanillaComm
  - A communication infrastructure for distributed RDBMS

Architecture of VanillaCore (1/2)



Architecture of VanillaCore (2/2)

- Interfaces:
  - SQL
  - JDBC
  - Native query interface
  - Storage interface (for file access)
- Key components:
  - Sever and infrastructures (jdbc, sql, tx, and utils)
  - Query engine
  - Storage engine

Outline

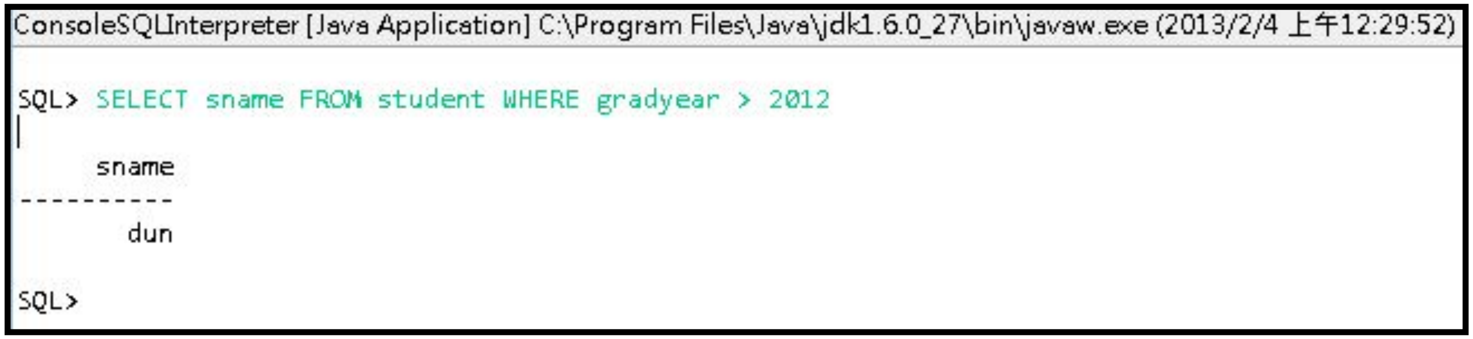
- Architecture of an RDBMS
- Query interfaces
  - SQL, JDBC, and Native
- Storage interface
  - RecordFile and metadata

The SQL Interface

- ***SQL*** (Structured Query Language) is a standardized interface
  - SQL-92, SQL-99, and later versions

Issuing SQL Commands

- Client-server mode:
  - Manual commands through `util.ConsoleSQLInterpreter`



```
ConsoleSQLInterpreter [Java Application] C:\Program Files\Java\jdk1.6.0_27\bin\javaw.exe (2013/2/4 上午12:29:52)

SQL> SELECT sname FROM student WHERE gradyear > 2012

     sname
----------
       dun

SQL>
```

  - Or in client programs through the ***JDBC*** interface
- Embedded mode:
  - Through the native query interface

Supported SQL Commands (1/5)

- VanillaCore supports a tiny subset of SQL-92
  - DDL: CREATE <TABLE | VIEW | INDEX>
  - DML: SELECT, UPDATE, INSERT, DELETE
- Limitations:
  - Types: int, long, double, and varchar
  - Single SELECT-FROM-WHERE block
    - No * in SELECT clause, no AS in FROM, no null value, no explicit JOIN or OUTER JOIN, only AND in WHERE, no parentheses, no computed value
  - Arithmetic expression only in UPDATE
  - No query in INSERT

Supported SQL Commands (2/5)

```
<Field>   := IdTok
<Constant>   := StrTok | NumericTok
<Expression> := <Field> | <Constant>
<BinaryArithmeticExpression>  :=
      ADD(<Expression>, <Expression>) |
SUB(<Expression>, <Expression>) |
MUL(<Expression>, <Expression>) |
DIV(<Expression>, <Expression>)
<Term>    := <Expression> = <Expression> |
<Expression> > <Expression>  |            <Expression> >=
<Expression> |            <Expression> < <Expression>  |
             <Expression> <= <Expression>
<Predicate>  := <Term> [ AND <Predicate> ]
```

Supported SQL Commands (3/5)

```
<Query>   := SELECT <ProjectSet> FROM <TableSet>        [
WHERE <Predicate> ] [ GROUP BY <IdSet> ]     [ ORDER BY
<SortList> [ DESC | ASC ] ]

<IdSet>   := <Field> [ , <IdSet> ]
<TableSet>   := IdTok [ , <TableSet> ]
<AggFn>   := AVG(<Field>) | COUNT(<Field>) |
COUNT(DISTINCT <Field>) | MAX(<Field>) |
MIN(<Field>) | SUM(<Field>)
<ProjectSet> := <Field> | <AggFn> [ , <ProjectSet>]
<SortList>   := <Field> | <AggFn> [ , <SortList>]
```

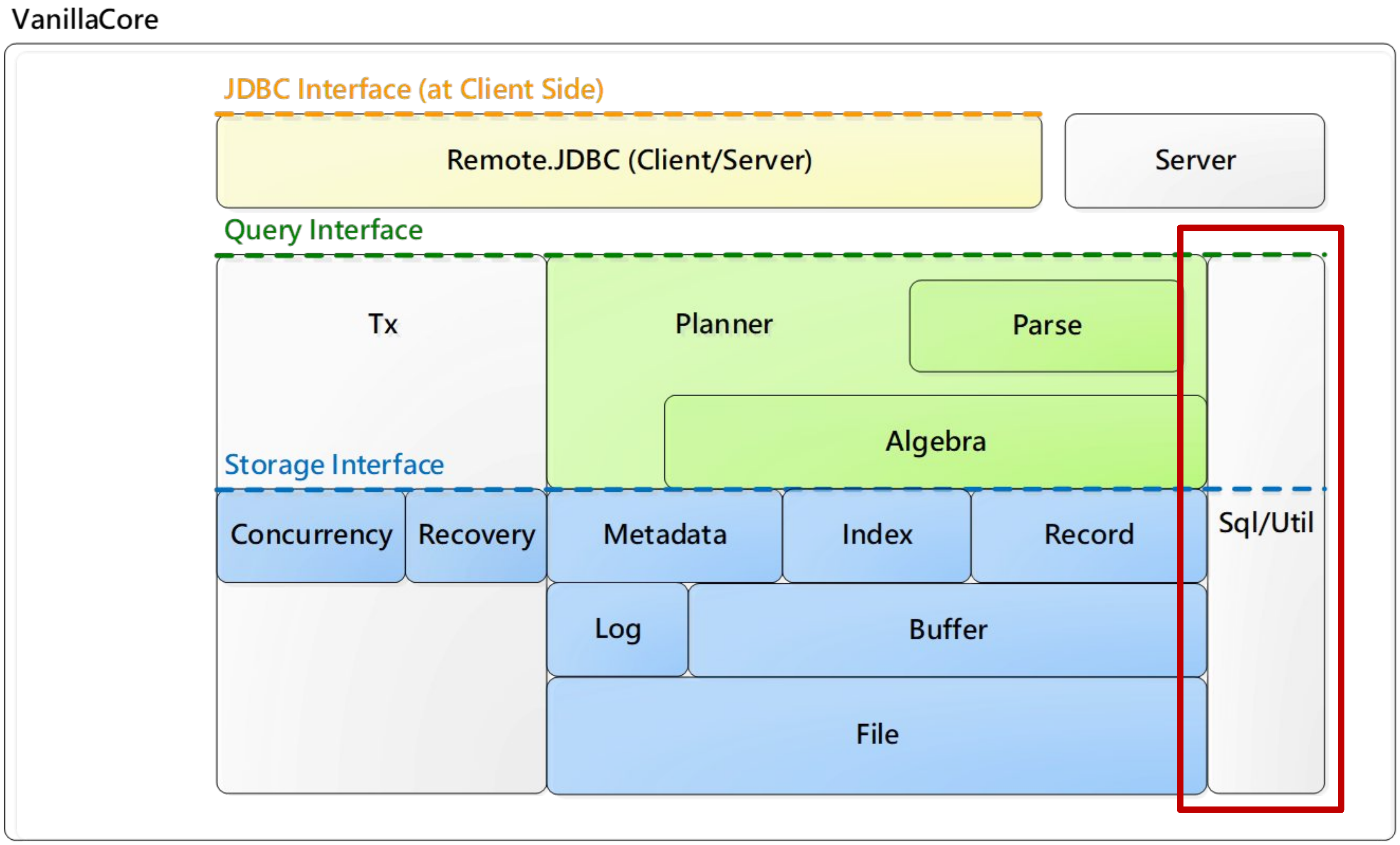
Supported SQL Commands (4/5)

```
<UpdateCmd>  := <Insert> | <Delete> | <Modify> | <Create>
<Create>  := <CreateTable> | <CreateView> |
<CreateIndex>
<Insert>  := INSERT INTO IdTok ( <FieldList> ) VALUES
       ( <ConstantList> )
<FieldList>  := <Field> [ , <Field> ]
<ConstantList>:= <Constant> [ , <Constant> ]
<Delete>  := DELETE FROM IdTok [ WHERE <Predicate> ]
<Modify>  := UPDATE IdTok SET <ModifyTermList>
    [ WHERE <Predicate> ]
```

Supported SQL Commands (5/5)

```
<ModifyExpression>  := <Expression> |
   <BinaryArithmeticExpression>
<ModifyTermList> := <Field> = <ModifyExpression>
      [ , <ModifyTermList> ]

<CreateTable>    := CREATE TABLE IdTok ( <FieldDefs> )
<FieldDefs>      := <FieldDef> [ , <FieldDef> ]
<FieldDef>       := IdTock <TypeDef>
<TypeDef>     := INT | LONG | DOUBLE |
VARCHAR ( NumericTok )
<CreateView>     := CREATE VIEW IdTok AS <Query>
<CreateIndex>    := CREATE INDEX IdTok ON IdTok
      ( <Field> )
```

Architecture of VanillaCore

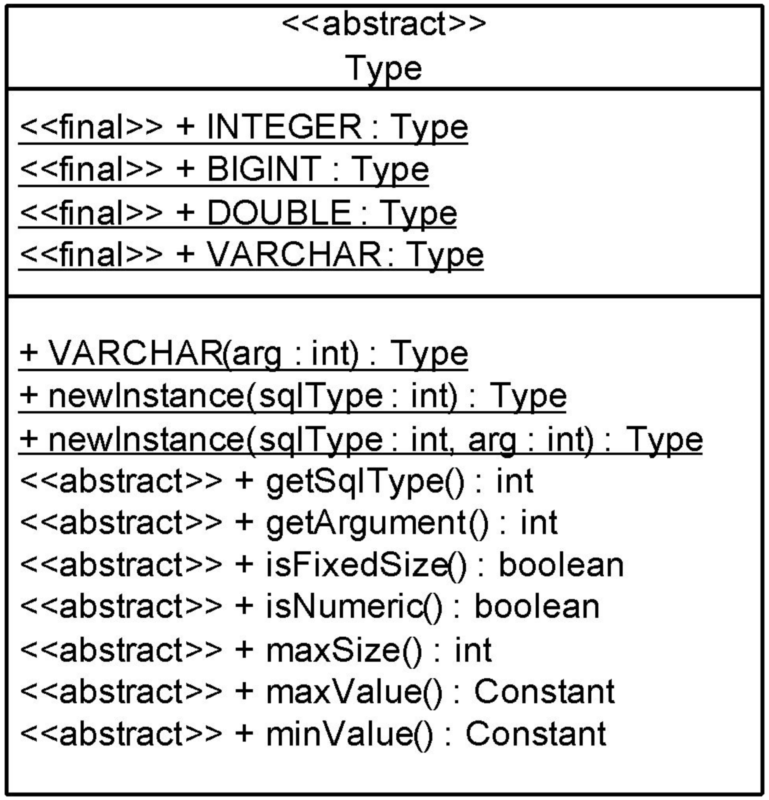


Utility Classes for SQL

- In `sql` package
- Types:
  - Numeric: `IntegerType`, `BigIntType`, and `DoubleType`
  - String: `VarcharType`
- Constants:
  - `IntegerConstant`, `BigIntConstant`, `DoubleConstant`, and `VarcharConstant`
- For relations:
  - `Schema`, `Record`
- For commands:
  - `Predicate`, `AggFn`

Types

- Each `Type` impl. denotes a supported SQL type

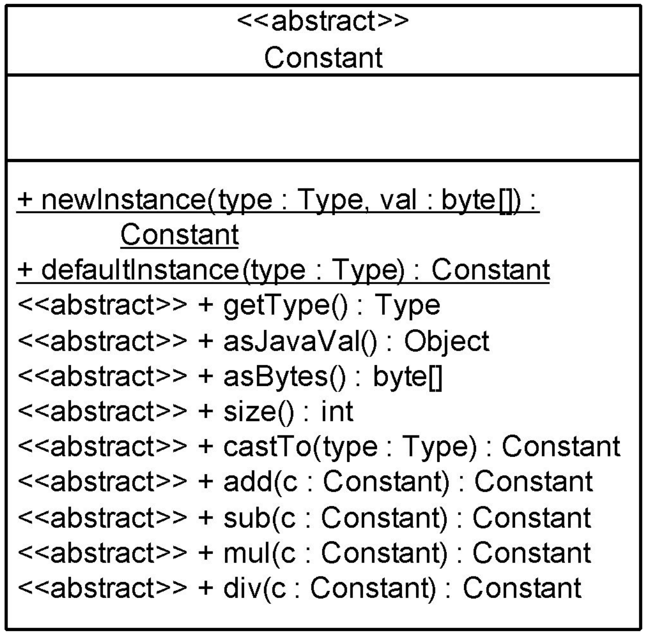


```
<<abstract>>
Type
-----------------------------------------------
<<final>> + INTEGER : Type
<<final>> + BIGINT : Type
<<final>> + DOUBLE : Type
<<final>> + VARCHAR: Type
-----------------------------------------------
+ VARCHAR(arg : int) : Type
+ newInstance(sqlType : int) : Type
+ newInstance(sqlType : int, arg : int) : Type
<<abstract>> + getSqlType() : int
<<abstract>> + getArgument() : int
<<abstract>> + isFixedSize() : boolean
<<abstract>> + isNumeric() : boolean
<<abstract>> + maxSize() : int
<<abstract>> + maxValue() : Constant
<<abstract>> + minValue() : Constant
```

java.sql.Types	vanilladb.sql.Type
INTEGER	IntegerType
BIGINT	BigIntType
DOUBLE	DoubleType
VARCHAR	VarcharType

Constants

- Each `Constant` impl. denotes a value of a supported type
  - Immutable
  - Arithmetics with auto type-upgrade



```
<<abstract>>
Constant

+ newInstance(type : Type, val : byte[]) :
        Constant
+ defaultInstance(type : Type) : Constant
<<abstract>> + getType() : Type
<<abstract>> + asJavaVal() : Object
<<abstract>> + asBytes() : byte[]
<<abstract>> + size() : int
<<abstract>> + castTo(type : Type) : Constant
<<abstract>> + add(c : Constant) : Constant
<<abstract>> + sub(c : Constant) : Constant
<<abstract>> + mul(c : Constant) : Constant
<<abstract>> + div(c : Constant) : Constant
```

vanilladb.sql.Type	Value type in Java
IntegerType	Integer
BigIntType	Long
DoubleType	Double
VarcharType	String

Relations

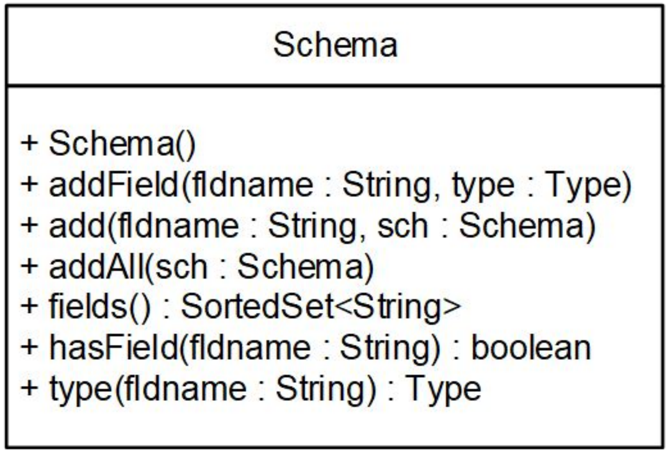
blog_pages

blog_id	url	created	author_id
33981	ms.com/…	2012/10/31	729
33982	apache.org/…	2012/11/15	4412

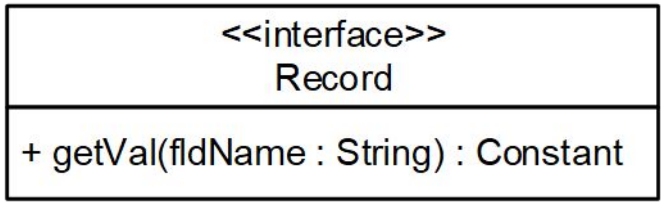
Schema

Record

Schema & Record


Schema
+ Schema()
+ addField(fldname : String, type : Type)
+ add(fldname : String, sch : Schema)
+ addAll(sch : Schema)
+ fields() : SortedSet<String>
+ hasField(fldname : String) : boolean
+ type(fldname : String) : Type

- Contains the name and type of each field in a table



- A map from field names to constants

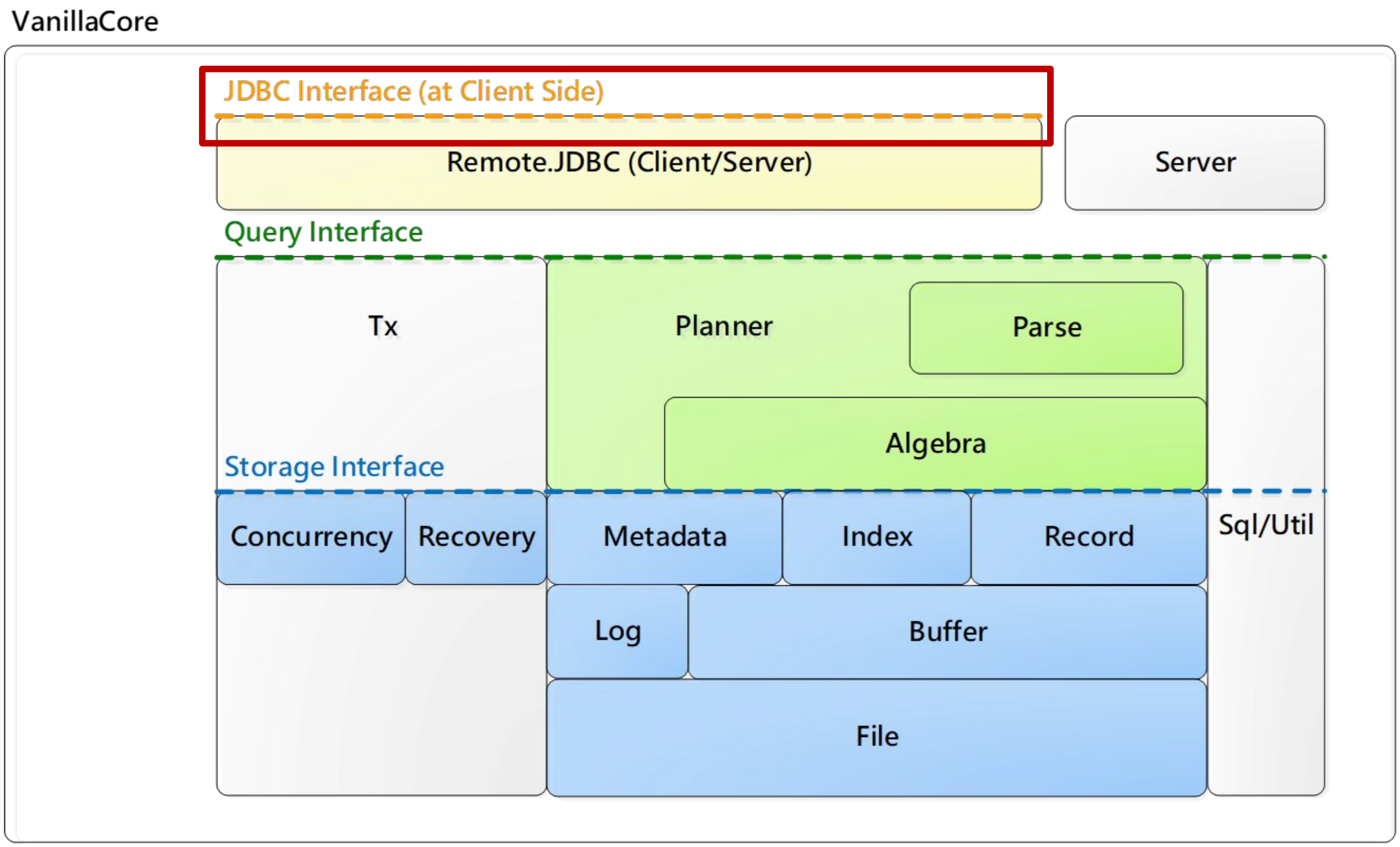
Commands

- Supporting WHERE: predicates in `sql.predicate` package
  - `Expression`, `FieldExpression`, `ConstantExpression`, `BinaryArithmeticExpression`, `Term`, and `Predicate`
- Supporting GROUP BY: aggregation functions in the `sql.aggfn` package
  - `AggregationFn`, `AvgFn`, `CountFn`, `DistictCountFn`, `MaxFn`, `MinFn` and `SumFn`

Outline

- Architecture of an RDBMS
- Query interfaces
  - SQL, JDBC, and native interface
- Storage interface
  - RecordFile and metadata

Architecture of VanillaCore (1/2)



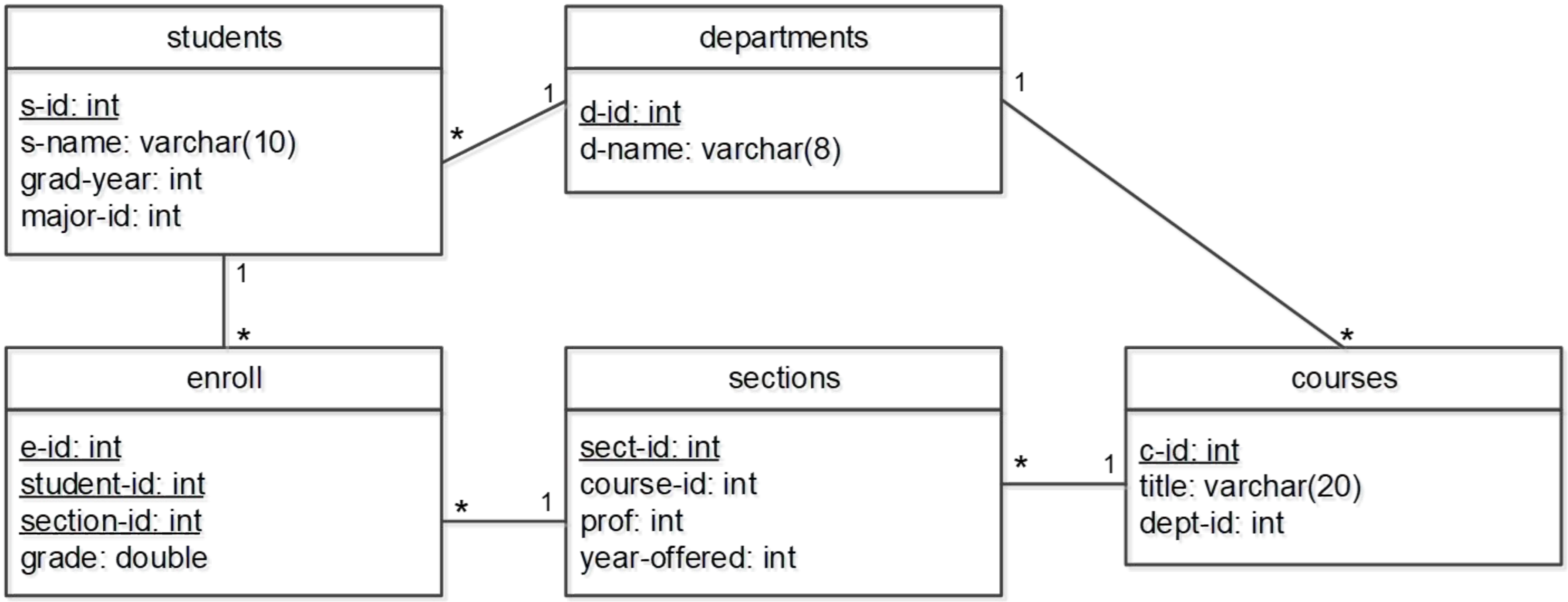
JDBC

- Defined in `java.sql`
- Java interfaces:
  - `Driver`, `Connection`, `Statement`, `ResultSet`, and `ResultSetMetaData`
- Implementation manages the transfer of data between a Java client and the RDBMS
- VanillaCore implements a tiny subset of JDBC
  - `org.vanilladb.core.remote.jdbc`

JDBC Programming

1. Connect to the server
2. Execute the desired query
3. Loop through the result set (for SELECT only)
4. ***Close*** the connection
    - A result set ties up valuable resources on the server, such as buffers and locks
    - Client should close its connection as soon as the database is no longer needed

Example: Finding Major

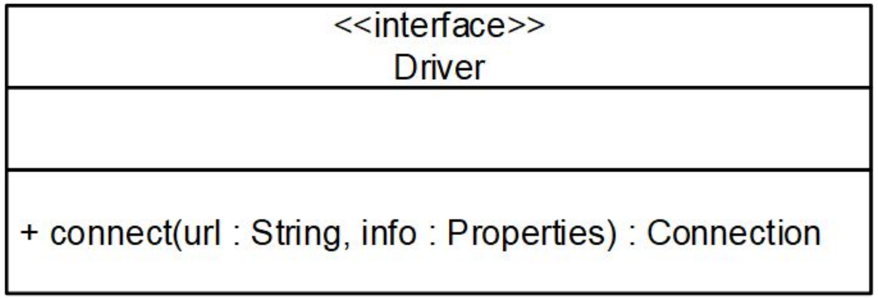


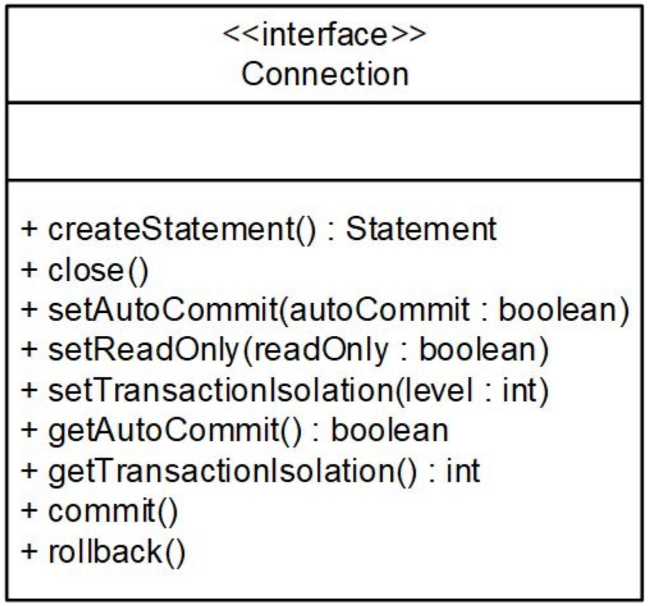
```
SELECT s-name, d-name
  FROM departments, students
  WHERE major-id = d-id
```

JDBC Program: Finding Major

```
Connection conn = null;
try {
      // Step 1: connect to database server
      Driver d = new JdbcDriver();
      conn = d.connect("jdbc:vanilladb://localhost", null);
      conn.setAutoCommit(false);
      conn.setReadOnly(true);
      // Step 2: execute the query
      Statement stmt = conn.createStatement();
      String qry = "SELECT s-name, d-name FROM departments, "
      + "students WHERE major-id = d-id";
      ResultSet rs = stmt.executeQuery(qry);
      // Step 3: loop through the result set
      rs.beforeFirst();
      System.out.println("name\tmajor");
      System.out.println("-------\t-------");
      while (rs.next()) {
            String sName = rs.getString("s-name");
            String dName = rs.getString("d-name");
            System.out.println(sName + "\t" + dName);
      }
      rs.close();
} catch (SQLException e) {
      e.printStackTrace();
} finally {
      try {
            // Step 4: close the connection
            if (conn != null)
            conn.close();
      } catch (SQLException e) {
            e.printStackTrace();
      }
}
```

API (1/2)

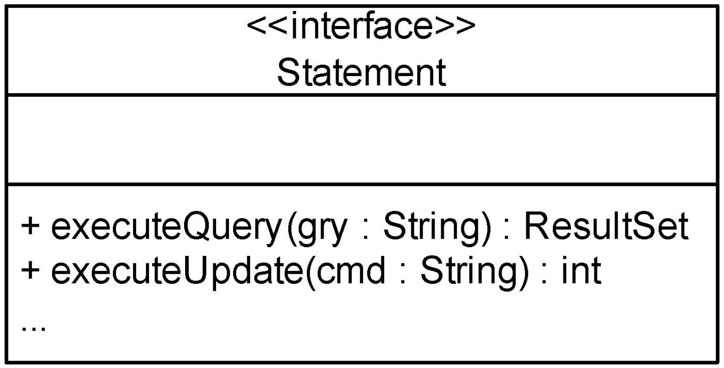




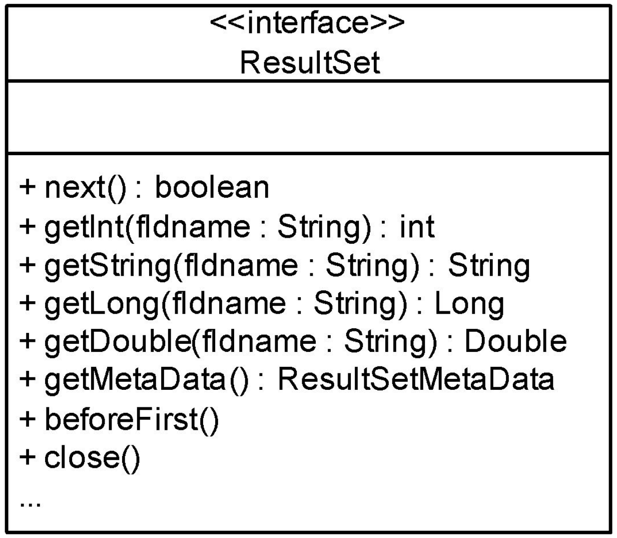
<<interface>>
Connection
+ createStatement() : Statement
+ close()
+ setAutoCommit(autoCommit : boolean)
+ setReadOnly(readOnly : boolean)
+ setTransactionIsolation(level : int)
+ getAutoCommit() : boolean
+ getTransactionIsolation() : int
+ commit()
+ rollback()

- A connection to the server

API (2/2)

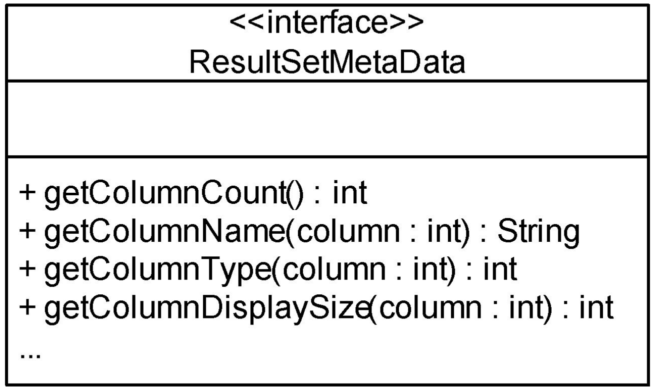


```
<<interface>>
Statement

+ executeQuery(gry : String) : ResultSet
+ executeUpdate(cmd : String) : int
...
```



```
<<interface>>
ResultSet

+ next() : boolean
+ getInt(fldname : String) : int
+ getString(fldname : String) : String
+ getLong(fldname : String) : Long
+ getDouble(fldname : String) : Double
+ getMetaData() : ResultSetMetaData
+ beforeFirst()
+ close()
...
```

- An iterator of output records

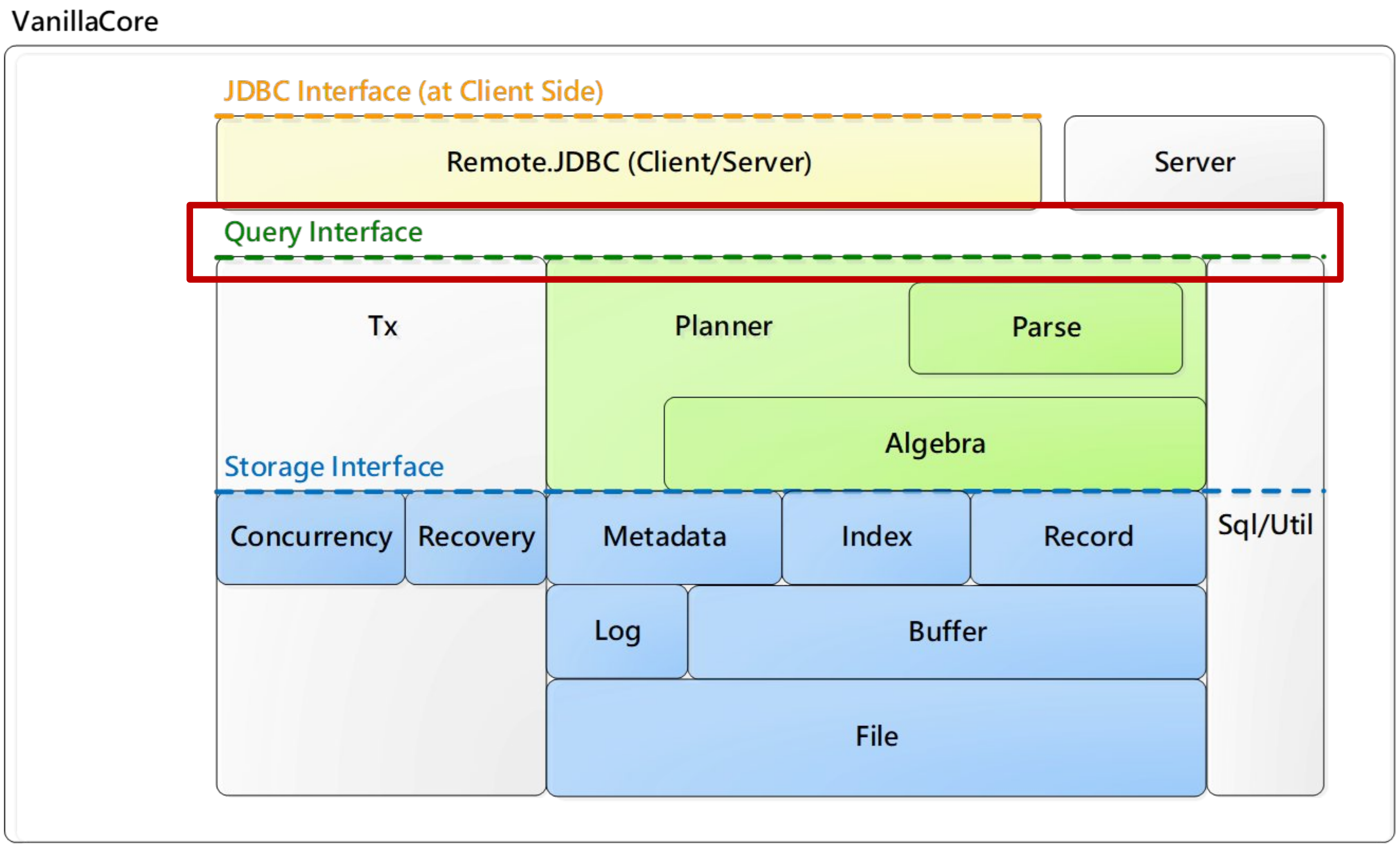


```
<<interface>>
ResultSetMetaData

+ getColumnCount() : int
+ getColumnName(column : int) : String
+ getColumnType(column : int) : int
+ getColumnDisplaySize(column : int) : int
...
```

Outline

- Architecture of an RDBMS
- Query interfaces
  - SQL, JDBC, and native interface
- Storage interface
  - RecordFile and metadata

Architecture of VanillaCore (1/2)



Native Program: Finding Major

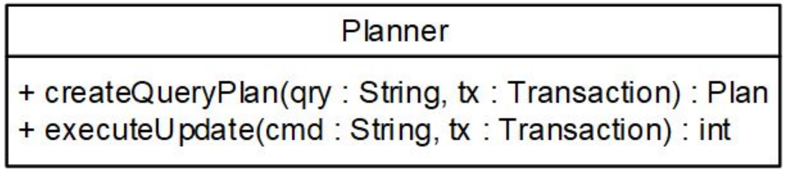
- JDBC client

```
Connection conn = null;
try {
        // Step 1: connect to database server
        Driver d = new JdbcDriver();
        conn = d.connect("jdbc:vanilladb://localhost", null);
        conn.setAutoCommit(false);
        conn.setReadOnly(true);
        // Step 2: execute the query
        Statement stmt = conn.createStatement();
        String qry = "SELECT s-name, d-name FROM departments, "
                + "students WHERE major-id = d-id";
        ResultSet rs = stmt.executeQuery(qry);
        // Step 3: loop through the result set
        rs.beforeFirst();
        System.out.println("name\tmajor");
        System.out.println("-------\t-------");
        while (rs.next()) {
                String sName = rs.getString("s-name");
                String dName = rs.getString("d-name");
                System.out.println(sName + "\t" + dName);
        }
        rs.close();
} catch (SQLException e) {
        e.printStackTrace();
} finally {
        try {
                // Step 4: close the connection
                if (conn != null)
                conn.close();
        } catch (SQLException e) {
                e.printStackTrace();
        }
}
```

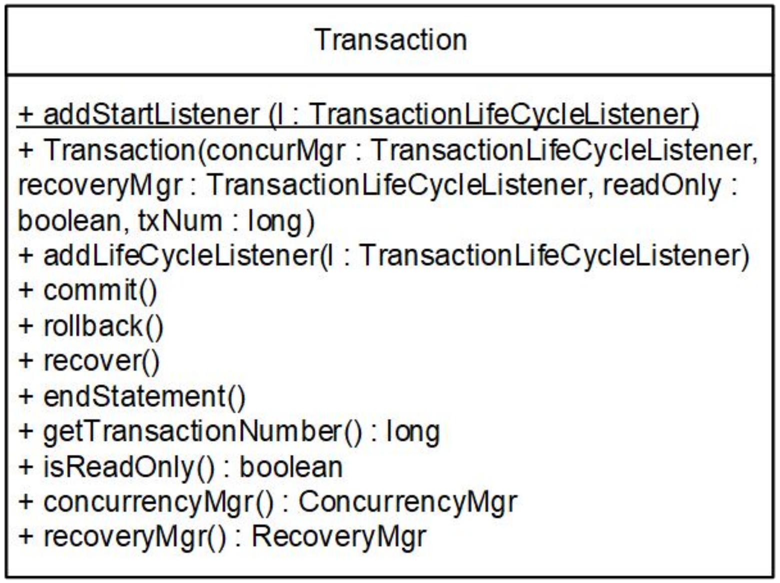
- Native (server side)

```
VanillaDb.init("studentdb");

// Step 1 correspondence
Transaction tx = VanillaDb.txMgr().newTransaction(
        Connection.TRANSACTION_SERIALIZABLE, true);

// Step 2 correspondence
Planner planner = VanillaDb.newPlanner();
String query = "SELECT s-name, d-name FROM departments, "
        + "students WHERE major-id = d-id";
Plan plan = planner.createQueryPlan(query, tx);
Scan scan = plan.open();

// Step 3 correspondence
System.out.println("name\tmajor");
System.out.println("-------\t-------");
while (scan.next()) {
        String sName = (String) scan.getVal("s-name").asJavaVal();
        String dName = (String) scan.getVal("d-name").asJavaVal();
        System.out.println(sName + "\t" + dName);
}
scan.close();

// Step 4 correspondence
tx.commit();
```

Planner & Transaction



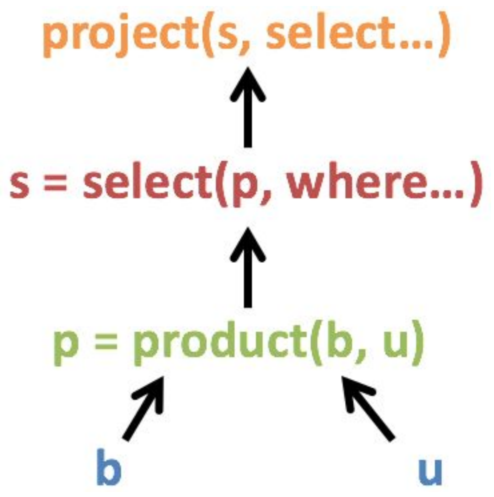
```
Planner
+ createQueryPlan(qry : String, tx : Transaction) : Plan
+ executeUpdate(cmd : String, tx : Transaction) : int
```

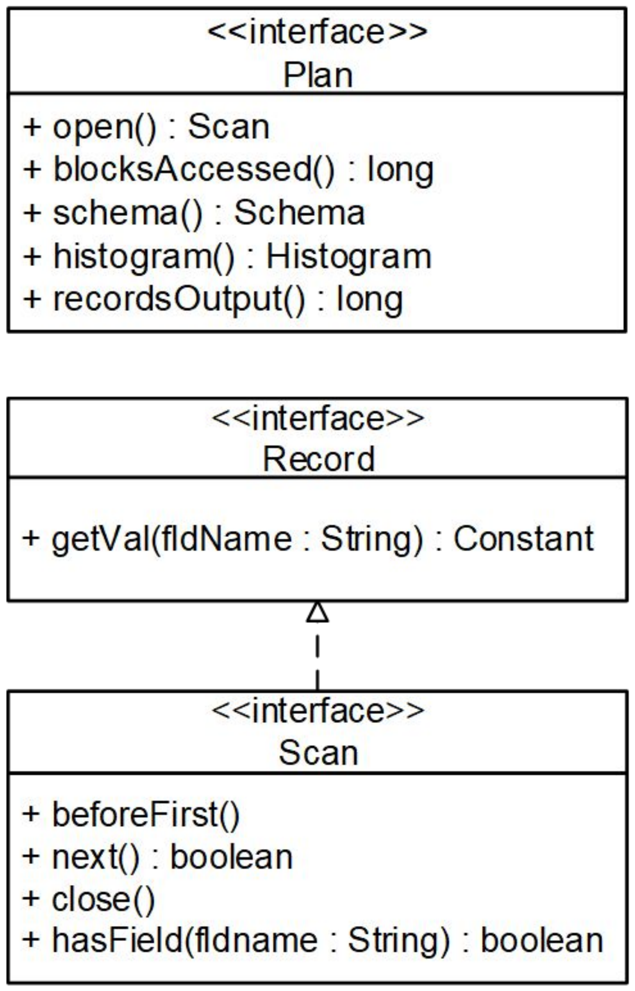


```
Transaction
+ addStartListener (l : TransactionLifeCycleListener)
+ Transaction(concurMgr : TransactionLifeCycleListener,
recoveryMgr : TransactionLifeCycleListener, readOnly :
boolean, txNum : long)
+ addLifeCycleListener(l : TransactionLifeCycleListener)
+ commit()
+ rollback()
+ recover()
+ endStatement()
+ getTransactionNumber() : long
+ isReadOnly() : boolean
+ concurrencyMgr() : ConcurrencyMgr
+ recoveryMgr() : RecoveryMgr
```

- All operations resulted from a planner are bound by the associated tx
- Planner: how to execute SQL efficiently?
- Transaction: how to ensure correctness?

Plans & Scans



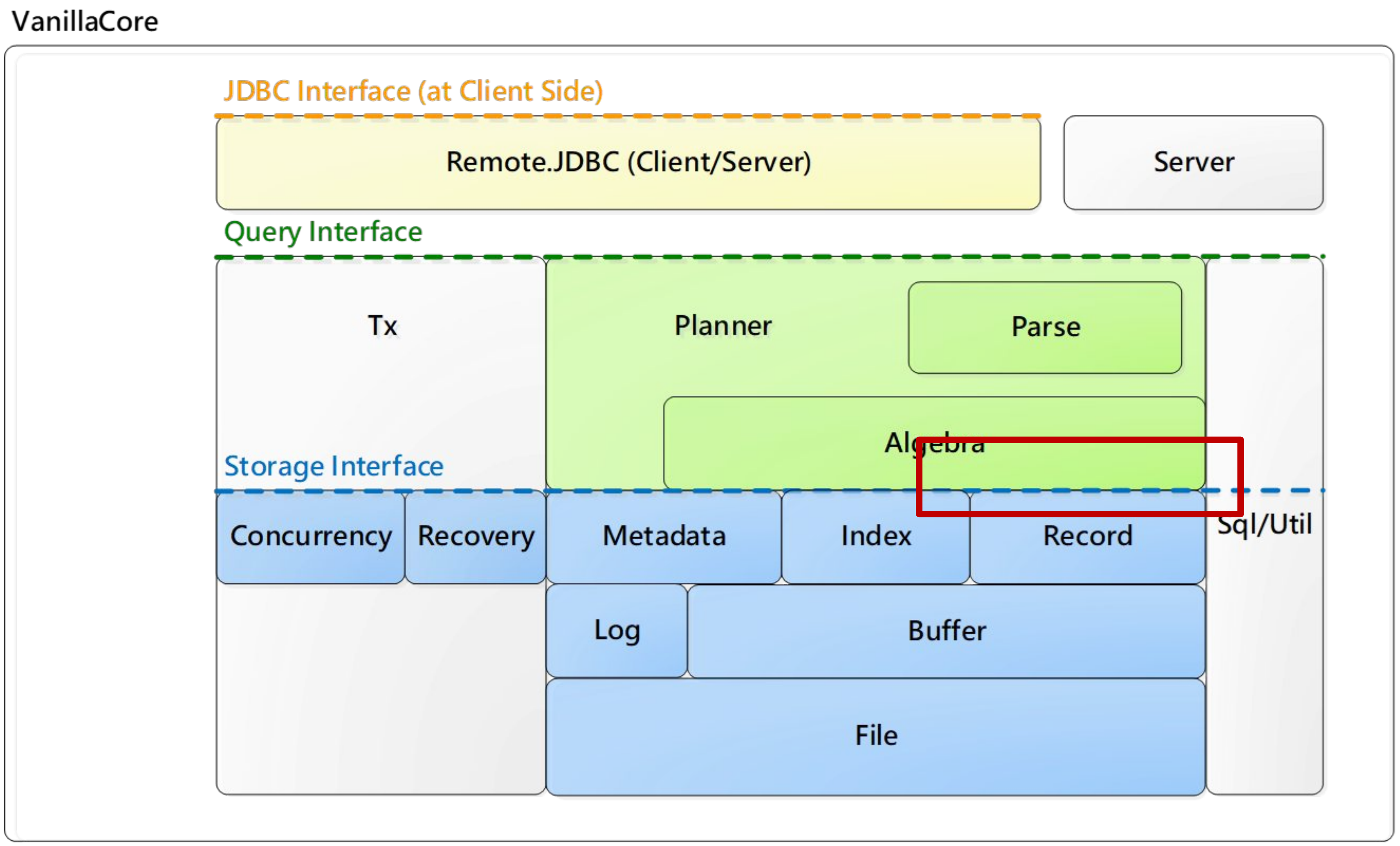


- SQL relational algebra
  - Each Plan obj. is a node in a plan tree
  - For ***cost estimation*** only
- `open()` returns a scan tree
  - Same architecture
  - Each node is an iterator of output records of a partial query (actual data access)

Outline

- Architecture of an RDBMS
- Query interfaces
  - SQL, JDBC, and native interface
- Storage interface
  - RecordFile and metadata

Architecture of VanillaCore (1/2)

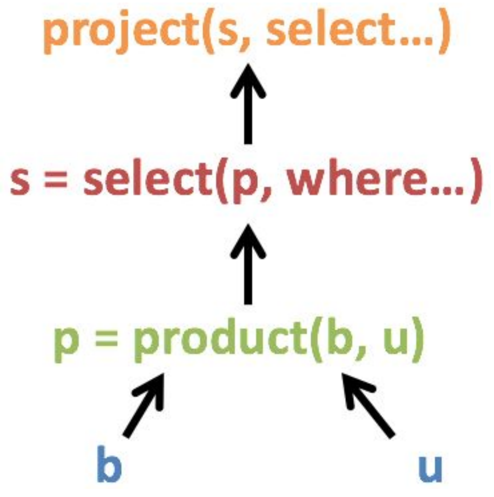


How are the databases/tables/records stored in a file system?

- Managed by the storage engine, e.g.,
  - Database: directory
  - Table: file
  - Record: bytes

How are they used in the query processing?

File Access



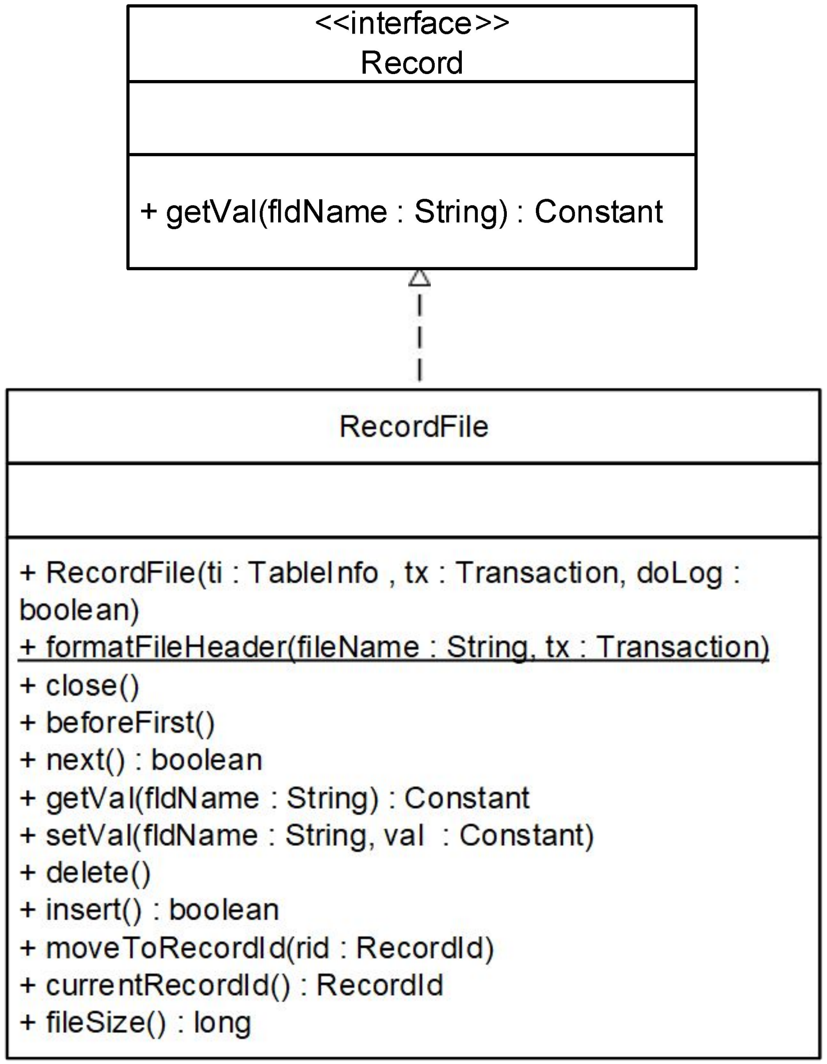
- Notice that the inputs of the lowest plans in a plan tree are always raw tables
  - Abstracted by `TablePlan`
  - The corresponding `TableScan` is an iterator of all records in a table
  - Each `TableScan` instance wraps a `RecordFile` instance

Files, Blocks, and Pages

- Definition: A ***block*** is the minimal sequence of bytes the OS reads/writes from/to a file at a time
  - Hides the difference of sectors in different devices
- Must be read into a ***page*** in memory first
  - Multiple writes to a page can be reflected to file at once using the system call `flush()`

Record File (1/2)

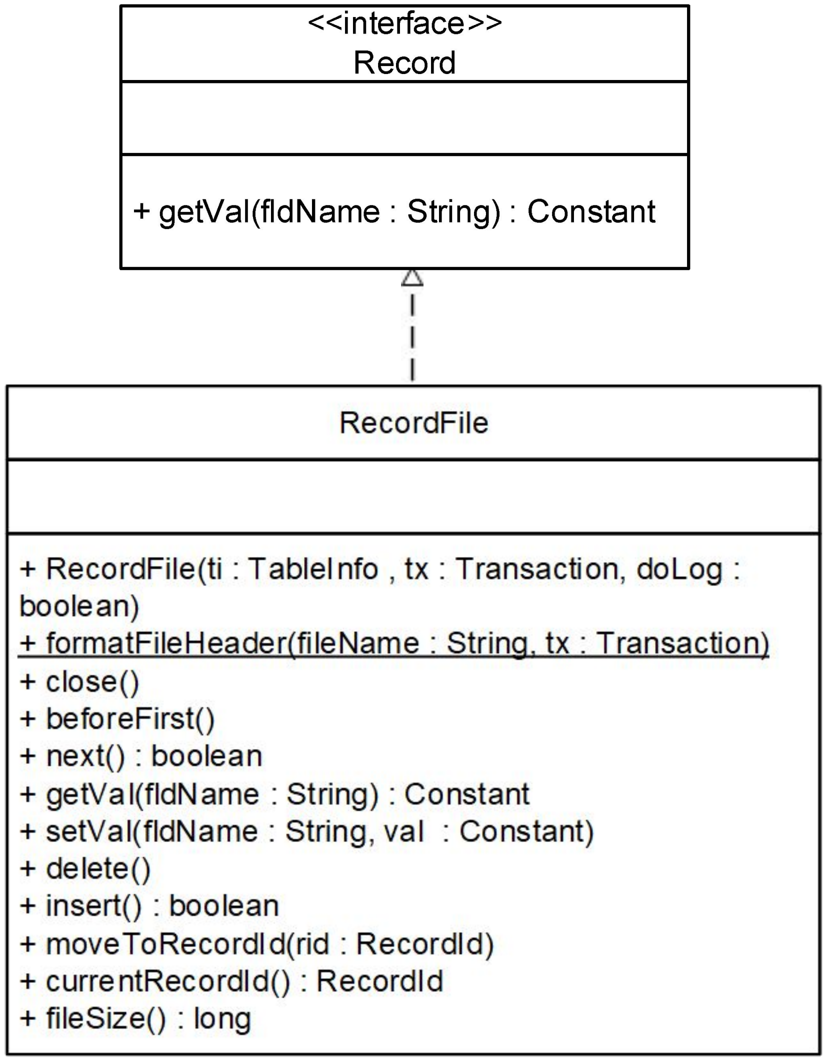
- Provides both random- and sequential-access methods to a file
- Random access: `moveToRecordId()`
  - RID = BID + shift-in-block
  - BID = file name + shift-in-file
- RecordFile is itself an iterator of a collection of records



<<interface>>
Record
+ getVal(fldName : String) : Constant
RecordFile
+ RecordFile(ti : TableInfo , tx : Transaction, doLog : boolean)
+ formatFileHeader(fileName : String, tx : Transaction)
+ close()
+ beforeFirst()
+ next() : boolean
+ getVal(fldName : String) : Constant
+ setVal(fldName : String, val : Constant)
+ delete()
+ insert() : boolean
+ moveToRecordId(rid : RecordId)
+ currentRecordId() : RecordId
+ fileSize() : long

Record File (2/2)

- Handles the caching automatically
- Reads/writes one block at a time from underlying file
- `getVal()` and `setVal()` access the current record in current page corresponding to the current block
- Calling next() may flush the page

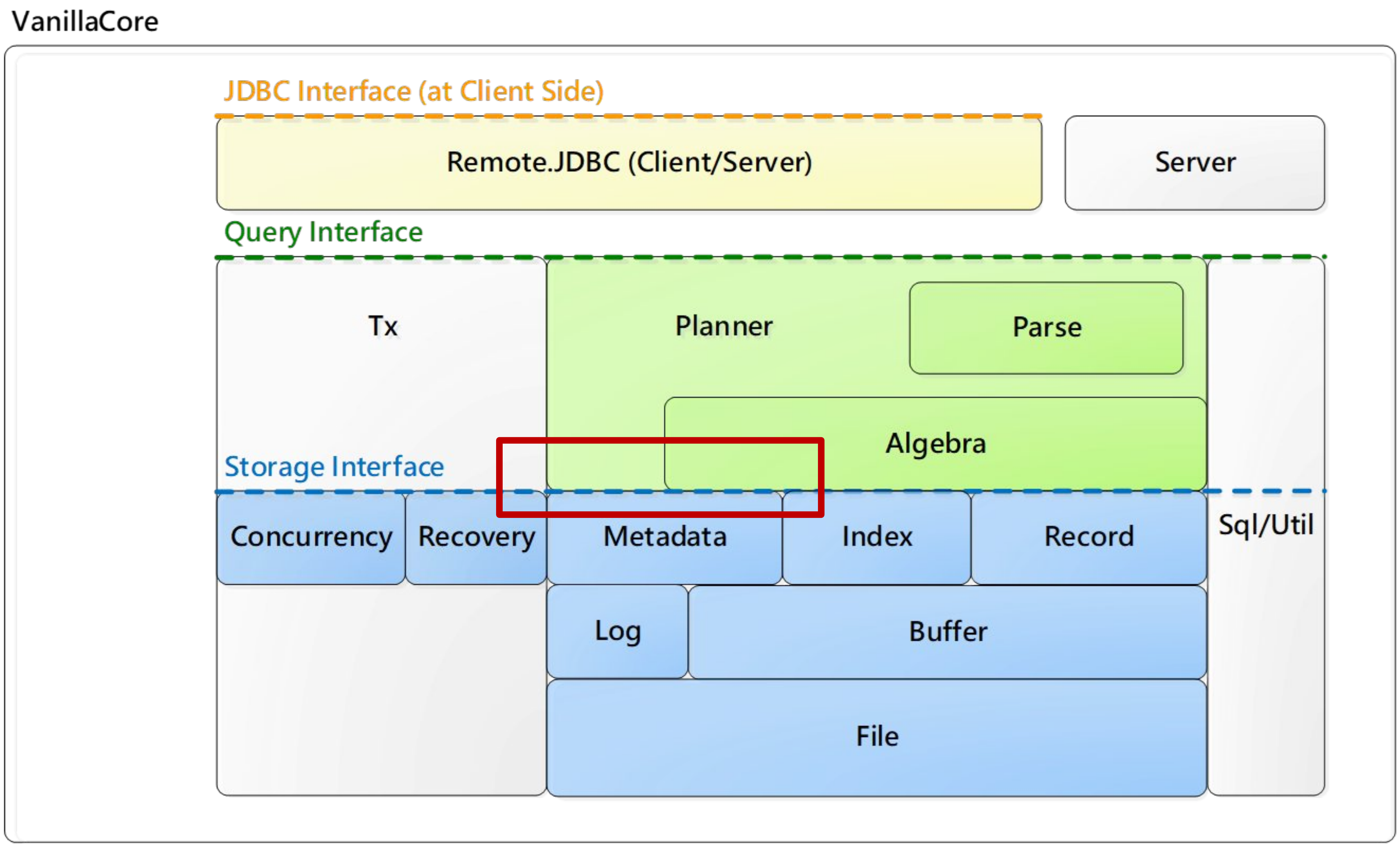


<<interface>>
Record
+ getVal(fldName : String) : Constant
RecordFile
+ RecordFile(ti : TableInfo , tx : Transaction, doLog : boolean)
+ formatFileHeader(fileName : String, tx : Transaction)
+ close()
+ beforeFirst()
+ next() : boolean
+ getVal(fldName : String) : Constant
+ setVal(fldName : String, val : Constant)
+ delete()
+ insert() : boolean
+ moveToRecordId(rid : RecordId)
+ currentRecordId() : RecordId
+ fileSize() : long

Outline

- Architecture of an RDBMS
- Query interfaces
  - SQL, JDBC, and Native
- Storage interface
  - RecordFile and metadata

Architecture of VanillaCore (1/2)



How does TableScan know which file to access for a table, and how many bytes for each record?

Metadata

- Definition: A ***metadata*** is the information about a database, apart from its contents.

Metadata in VanillaCore

- Table metadata
  - Describes the file of each table, and structure of the table's records such as the length, type, and offset of each field
- View metadata
  - Describes the properties of each view, such as its definition and creator
- Index metadata
  - Describes the indexes that have been defined on each field
- Statistical metadata
  - Describes the statistics of each table useful to estimating the cost of a plan

Metadata in Database System (1/2)

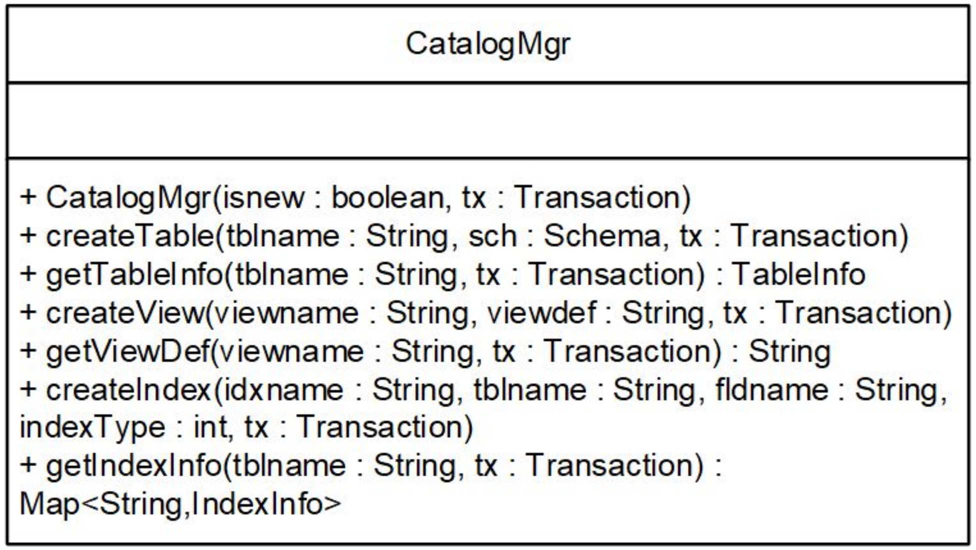
- VanillaCore stores the first three types of metadata in a collection of special tables called the ***catalog tables***
  - tblcat.tbl, fldcat.tbl, idxcat.tbl and viewcat.tbl
  - Updated each time when a table/view/index is created
- Why?
  - Allows the metadata to be queried like normal data

Metadata in Database System (2/2)

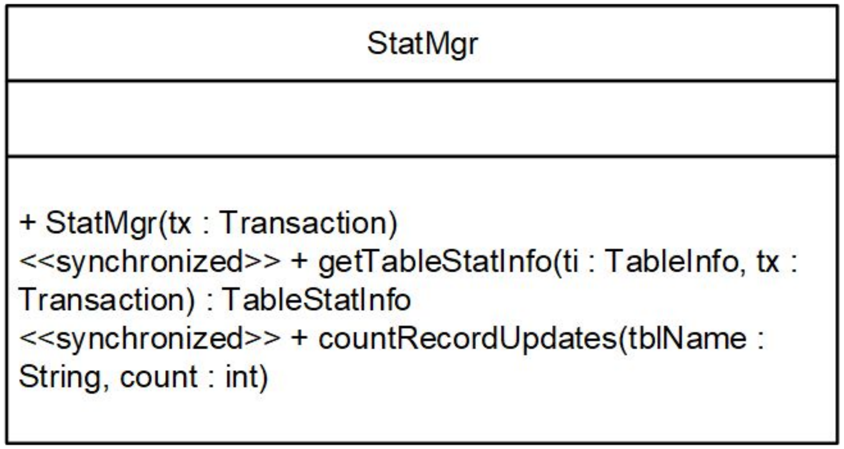
- Statistical metadata is kept in memory and updated periodically
- Why?
  - No need to be accurate
  - Accessed by every plan tree, must be very fast

Metadata Management

- The storage engine provides ***catalog manager*** and ***statistic manager***
  - It is the `Planner` that notifies `StatMgr` about the changes to a DB
- Related package
  - `org.vanilladb.core.storage.metadata`



```
CatalogMgr

+ CatalogMgr(isnew : boolean, tx : Transaction)
+ createTable(tblname : String, sch : Schema, tx : Transaction)
+ getTableInfo(tblname : String, tx : Transaction) : TableInfo
+ createView(viewname : String, viewdef : String, tx : Transaction)
+ getViewDef(viewname : String, tx : Transaction) : String
+ createIndex(idxname : String, tblname : String, fldname : String,
indexType : int, tx : Transaction)
+ getIndexInfo(tblname : String, tx : Transaction) :
Map<String,IndexInfo>
```



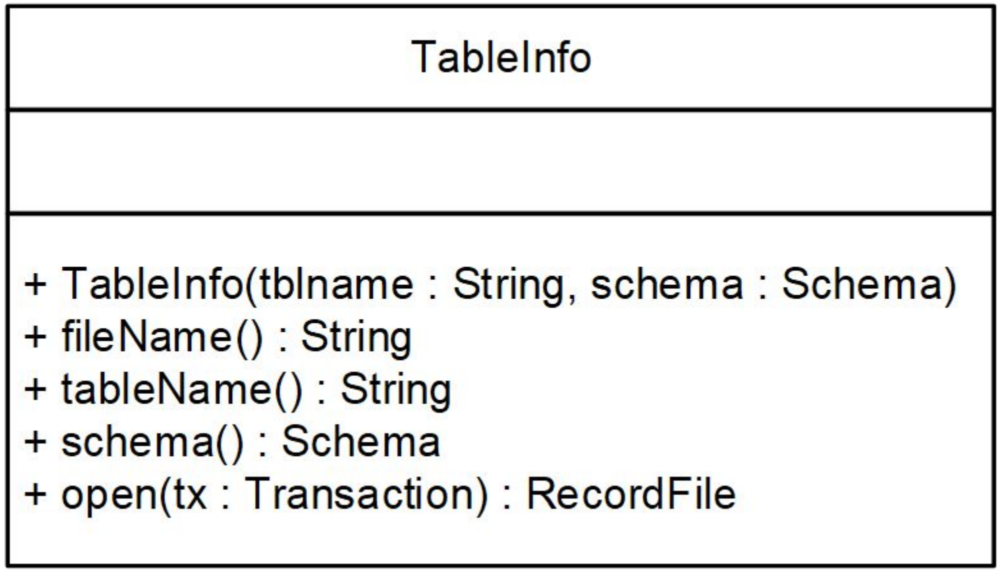
```
StatMgr

+ StatMgr(tx : Transaction)
<<synchronized>> + getTableStatInfo(ti : TableInfo, tx :
Transaction) : TableStatInfo
<<synchronized>> + countRecordUpdates(tblName :
String, count : int)
```

Using Table Metadata

- When creating a table, the `Planner` calls `CatalogMgr.createTable(tbln, sch, tx)`
  - Calculates and writes table metadata to catalog
- At the lowest level of a plan tree, the `TablePlan`/`Scan` can extract the metadata of the specified table through `CatalogMgr.getTableInfo(tbln, tx)`

Table Info.

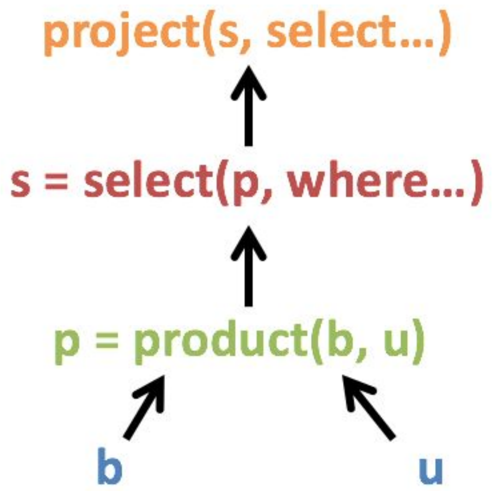
- `org.vanilladb.core.storage.metadata.TableInfo`


TableInfo
+ TableInfo(tblname : String, schema : Schema)
+ fileName() : String
+ tableName() : String
+ schema() : Schema
+ open(tx : Transaction) : RecordFile

Using the Table Metadata (Planner)

```java
VanillaDb.init("studentdb");
CatalogMgr catalogMgr = VanillaDb.catalogMgr();

// Create dept table
Transaction tx1 = VanillaDb.txMgr().newTransaction(
    Connection.TRANSACTION_SERIALIZABLE, false);

Schema sch = new Schema();
sch.addField("did", Type.INTEGER);
sch.addField("dname", Type.VARCHAR(20));
catalogMgr.createTable("dept", sch, tx1);

tx1.commit();
```

Using the Table Metadata (TablePlan/Scan)


$$\begin{tikzcd}
 & \text{project(s, select...)} & \\
 & \text{s = select(p, where...)} \arrow[u] & \\
 & \text{p = product(b, u)} \arrow[u] & \\
\text{b} \arrow[ur] & & \text{u} \arrow[ul]
\end{tikzcd}$$

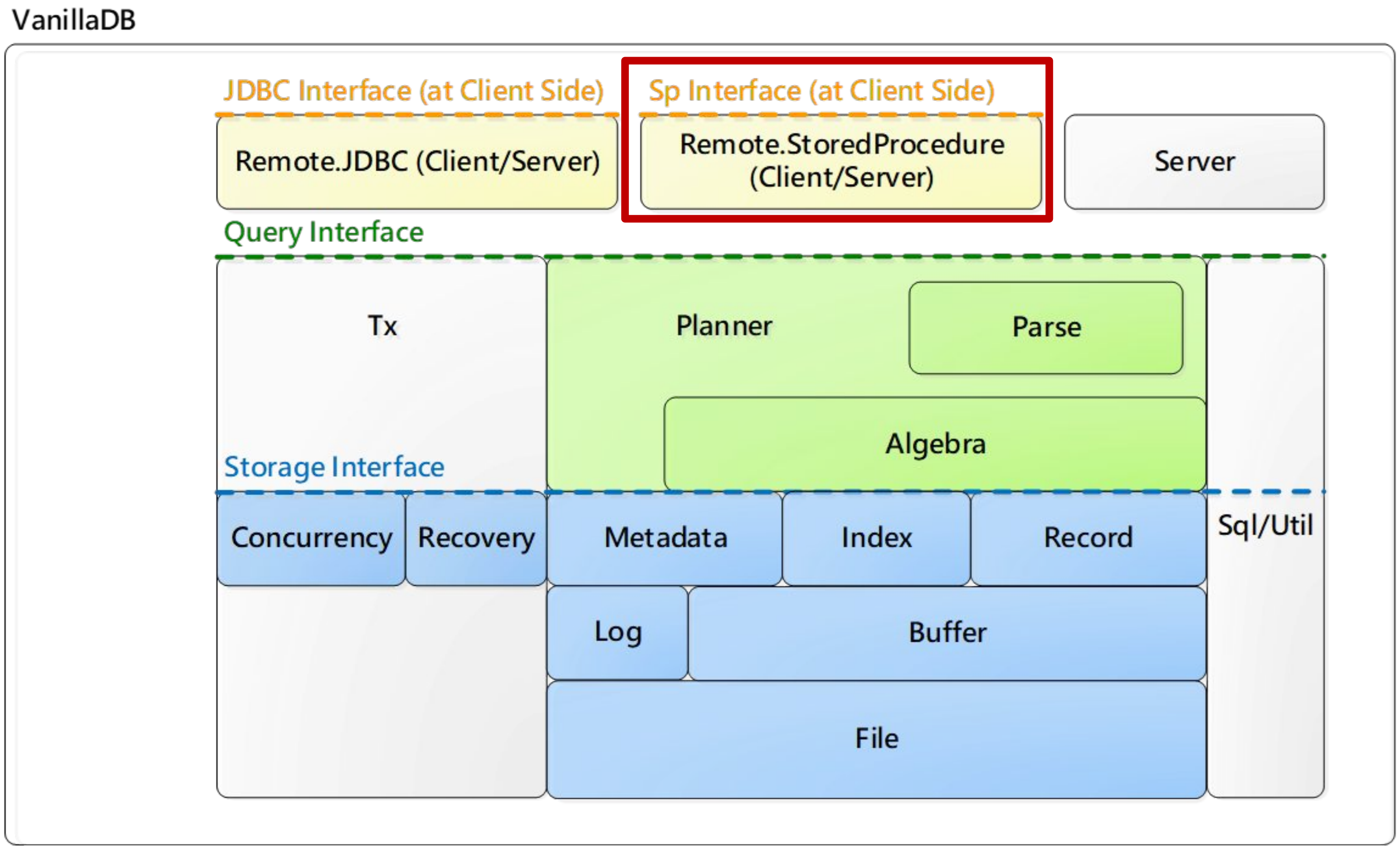
```
// Print the name of each department
Transaction tx2 = VanillaDb.txMgr().newTransaction(
    Connection.TRANSACTION_SERIALIZABLE, true);

TableInfo ti = catalogMgr.getTableInfo("dept", tx2);
RecordFile rf = ti.open(tx2);
rf.beforeFirst();
while (rf.next())
    System.out.println(rf.getVal("dname"));
rf.close();

tx2.commit();
```

References

- M.M. Astrahan et al., System R: relational approach to database management, *ACM Transactions on Database Systems*, Vol. 1, No. 2, 1976
- J. M. Hellerstein et al., Architecture of a database system, *Foundations and Trends in Databases*, Vol. 1, No. 2, 2007
- Edward Sciore, Chapters 8 & 20, *Database Design and Implementation*, 2008

Assignment (1/3)

- Actually, VanillaCore supports an additional client/server interface called ***stored procedures***



Assignment (2/3)

- In package remote.storedprocedure
- Trace the code yourself

Assignment (3/3)

- Given a JDBC client, rewrite it using the stored procedures
- Using the provided data population and benchmark tool to compare their performance