VOLTDB SQL COMPILATION

How To Turn Queries Into Plans.

created by bill.white@voltdb.com | 2018-05-22 | online | src

Note: Use the question mark key (?) for

SQL. SQL Query Syntax.

- Here we give a very short *reprise* of SQL query syntax, just to fix terminology.
- Tables and indexes are straightforward.
- Select statements have interesting structure.

SQL.1. A *Reprise* of SQL Query Syntax

Consider this query:

```
select max(t1.a * t2.a)              <-- display list
    from r1 as t1 join r1 as t2      <-- range variables
        on t1.a < 0 and t1.a = t2.a  <-- join conditions
    where t1.a < 100                 <-- where condition
    group by t1.a, t2.a              <-- group by expressions
    having sum(t1.a) > 0             <-- having condition
    order by t1.a                    <-- order by expression
```

A SQL query plan is basically a nested loop.

- For this SQL:

```
select * from  outerTable as O join innerTable as I
    on O.id = I.id;
```

The execution is a variant of this pseudo code:

Everything else is an optimization.

- We may scan an index rather than a table.
- We may filter the rows on one table alone, or on both.
- We may swap the outer and inner tables.

Plans. Execution Plans

- Before we describe the planner we have to explain what the desired output would be.
- Look at the JavaDoc comment for the package:

```
org.voltdb.planner
```

for much more detail than we can give in

Plans.1 An Example.

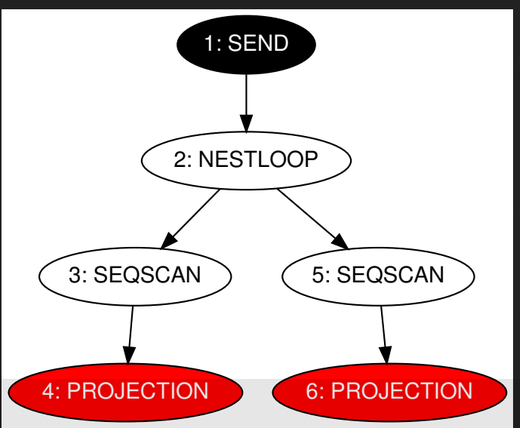
This SQL query:

```
select * from r join s on r.id = s.id;
```

Produces this plan tree (read from bottom to top),

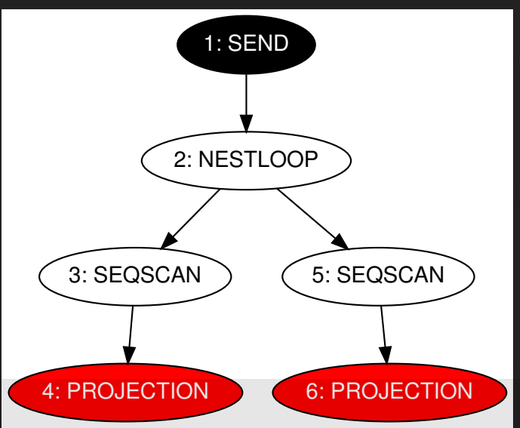
Plans.1.a. SEQSCAN Nodes.

- The White nodes are sequential scan nodes.
The output will be the the rows of the input table, but processed through the Red projection nodes.



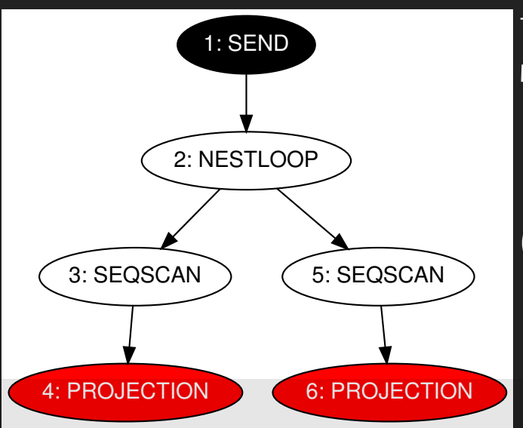
Plans.1.b. PROJECTION Nodes.

- The Red nodes are projection nodes, which select columns.
They can also compute expressions.
These are red in this diagram, which denotes inline nodes.



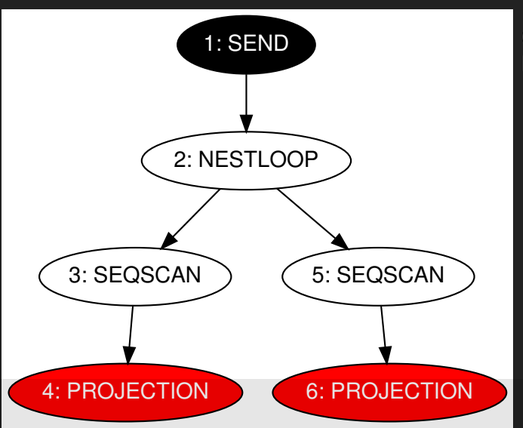
Plans.1.c. NESTLOOP Nodes.

- The NESTLOOP nodes compute all pairs <r1, r2> from r and s.
The join condition r.id = s.id is computed here, and rows are filtered out.



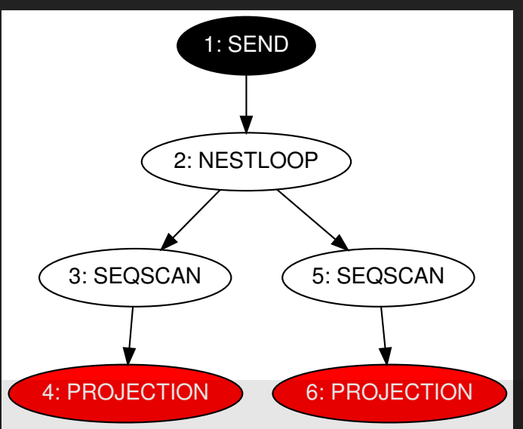
Plans.1.d. SEND Nodes.

- The SEND node packages up the table in a format which can be sent back to the server.



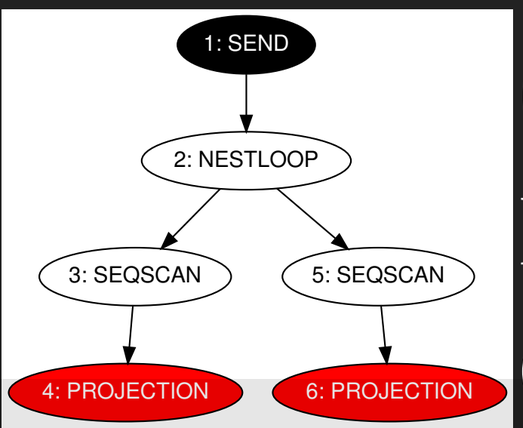
Plans.1.e. How to make this diagram.

- In a shell window,
  - run export VOLTDB_OPTS="-Dorg.voltdb.compilerdebug=true"
  - Start a voltdb server *in this shell window*.



Plans.1.f. How to view the diagram.

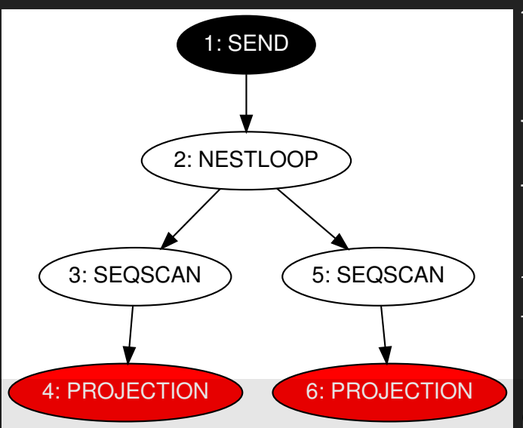
- In the directory in which the server is running, look for the directory named debugoutput/statement-all-plans there will be a file named WINNER-0.dot. Hunt around for it.
- Process this file with the graphviz tool dot, like this:



Plans.1.g. debugoutput.

There are many other files in debugoutput which are interesting.

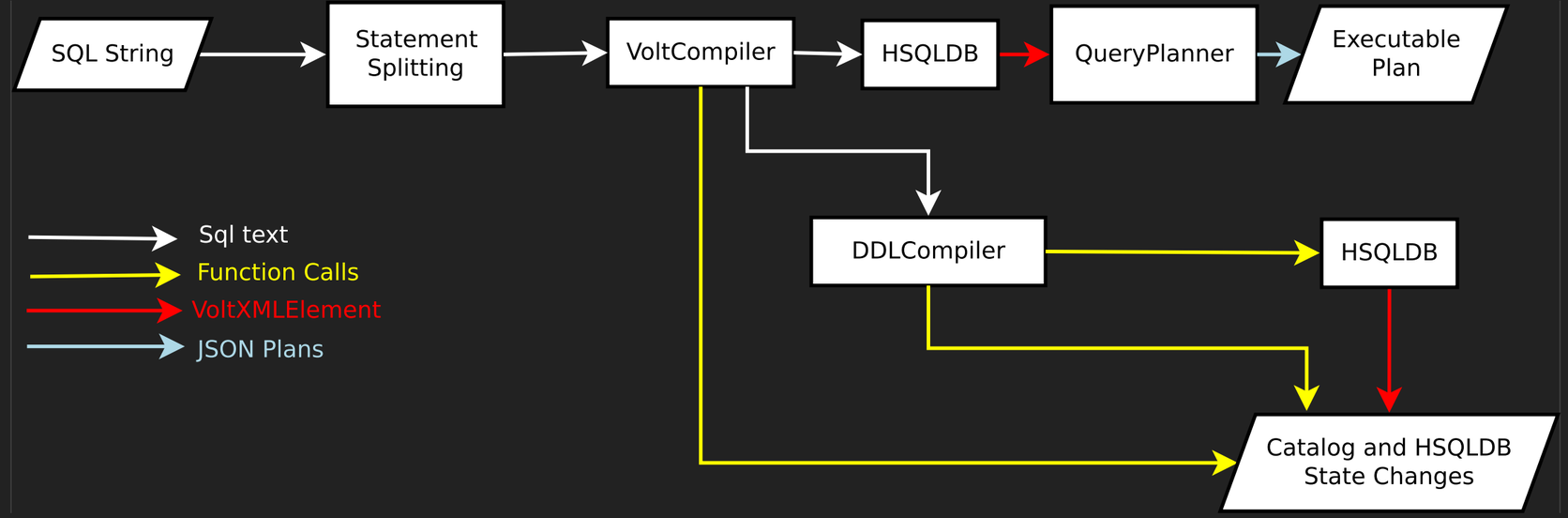
- The actual JSON text for the plan is there someplace.
HSQLDB's notion of the catalog schema is in schema-xml/hsql-catalog-output.xml



Phases.1. The main phases.

- Statement Splitting
- VoltCompiler
- HSQL
  - Output 1: Catalog and HSQLDB State Changes
  - Output 2: VoltXMLElement objects.
- QueryPlanner

Phases.1.1. Data Flow



Phases.2. A debugging tip.

- The method

```
org.hsqldb_voltpatches.HSQLInterface.getXMLCompiledStat
```

transforms a SQL statement string into a VoltXMLElement object.

- VoltXMLElement is one of the several intermediate forms we use. More on this

Phases.3. Logging

- In the file tests/log4j-allconsole.xml there is a commented-out element named HSQLDB_COMPILER.
- If you enable this, by editing the comment characters out, it will print the SQL, the HSQLDB intermediate form, and the VoltXMLElement for each SQL statement.
- This is often enormously useful.

Phases.3.1. A Warning.

- Don't forget to reinstate the comment in tests/log4j-allconsole.xml.
- Don't check this file in with logging enabled.
- It's possible to do this with a running server, but it's done slightly differently.

Split. Statement Splitting

- Split a string into SQL statements. We can only process one SQL statement at a time.
- This is complicated by the presence of literal strings, comments and multi-statement procedures.
- This is mostly in the member function:

Spit.a Semicolons in comments and strings.

- We strip comments here.
- We scan the string for semicolons, doing the right thing with multi-statement procedures and strings.

Spit.b Splitting for VMC.

- There is a similar routine in sqlcmd and the java script for VMC.

VoltCompile.1. VoltCompiler

- The class is org.voltdb.compiler.VoltCompiler.
- Used to process some kinds of VoltDB syntax which HSQL cannot handle.
  - create procedure ...
  - partition table ...
  - create function ...
- Mostly driven by regular expression

VoltCompile.2. More about VoltCompiler

- Many such statements never actually reach HSQL.
- Some regular expression matching is used even for HSQL-legal constructions to extract information that's hard to get from HSQL.
  - What table is being activated in a create index, alter table or create

HSQLDB.
HSQLDB
HSQLDB

- HSQLDB is the *Hypersonic SQL Database*.
- It is our SQL front end.
- It started out as version 1.9.3. The current HSQLDB version is 2.4.1.
- It really cannot be be upgraded. There are too many VoltDB specific changes.

HSQLDB.1.
HSQLDB parsing.

- It parses SQL text, using three big recursive descent parsers in
  - org.voltpatches_hsqldb.ParserDDL
  - org.voltpatches_hsqldb.ParserDML
  - org.voltpatches_hsqldb.ParserDQL

HSQLDB.2. HSQLDB DDL Parsing.

- For DDL it adds another internal representation of a table or index to its symbol table.
- We can extract VoltXMLElement from this symbol table.

HSQLDB.3. HSQLDB DML and DQL Parsing.

- Since DML and DQL don't define symbols - tables or indexes - we get an HSQLDB internal representation of a Statement.
- We can get VoltXMLElement from the

HSQLDB.4. An Idea.

- There is a function org.hsqldb_voltpatches.Statement.describe which describes HSQLDB's intermediate form for SQL statements.
- It really does a bad job of it. There are many NPEs and its indenting is lousy.
- It would be a good short-term project to fix this,

Note.1. Some future ideas.

Note.1.a. Refactoring

- It would be a very good idea to throw everything above this slide away.
  - VoltCompiler is extremely difficult to extend, and is the source of much teeth gnashing.
  - HSQLDB has peculiar error messages and annoying bugs.

Note.1.b. How we might refactor this all away.

- It would be a very good thing to write a proper SQL parser.
- For DML and DQL this new parser would take SQL strings and produce VoltXMLElement objects.

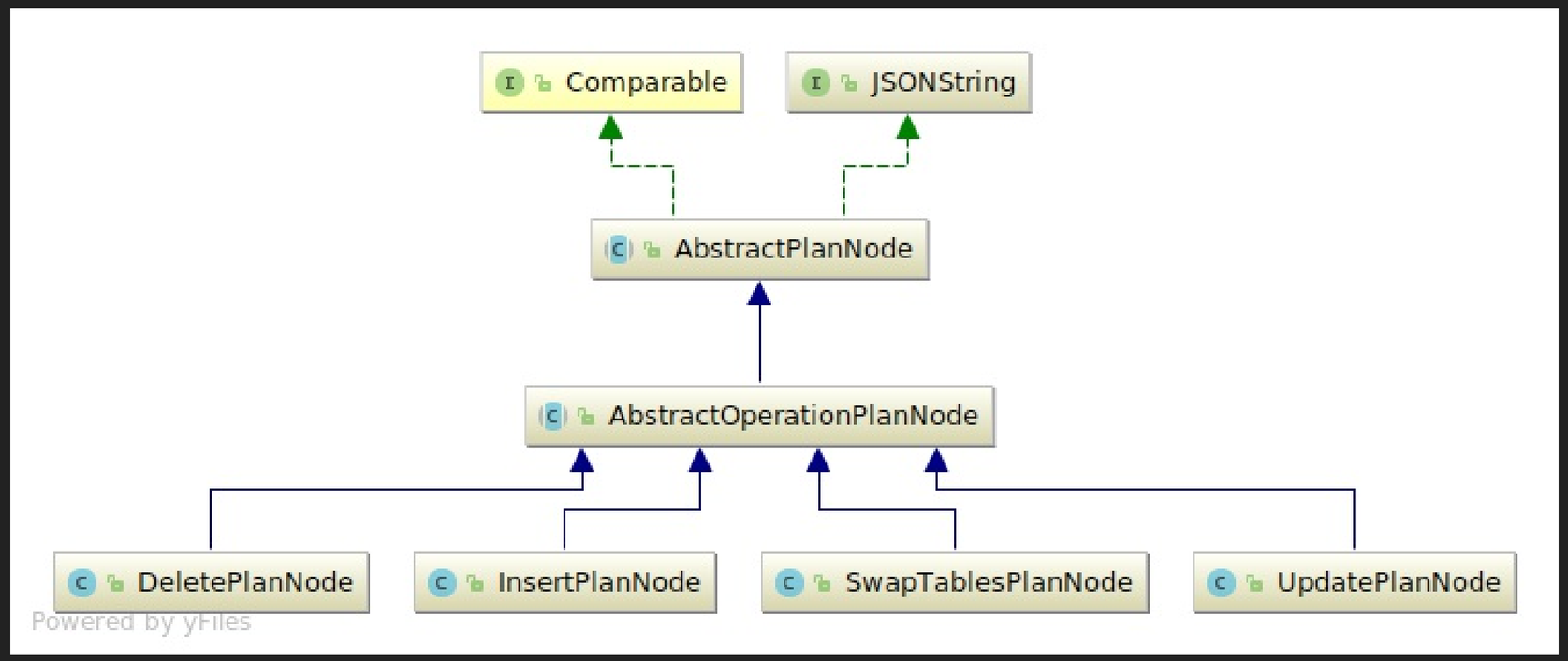
Plans.2. Plan Nodes

- A *Plan Node* is an abstract representation of a transformation on one or more tables.
- In the ExecutionEngine we represent a plan as a vector of something called *Executors*, which do the actual computations.

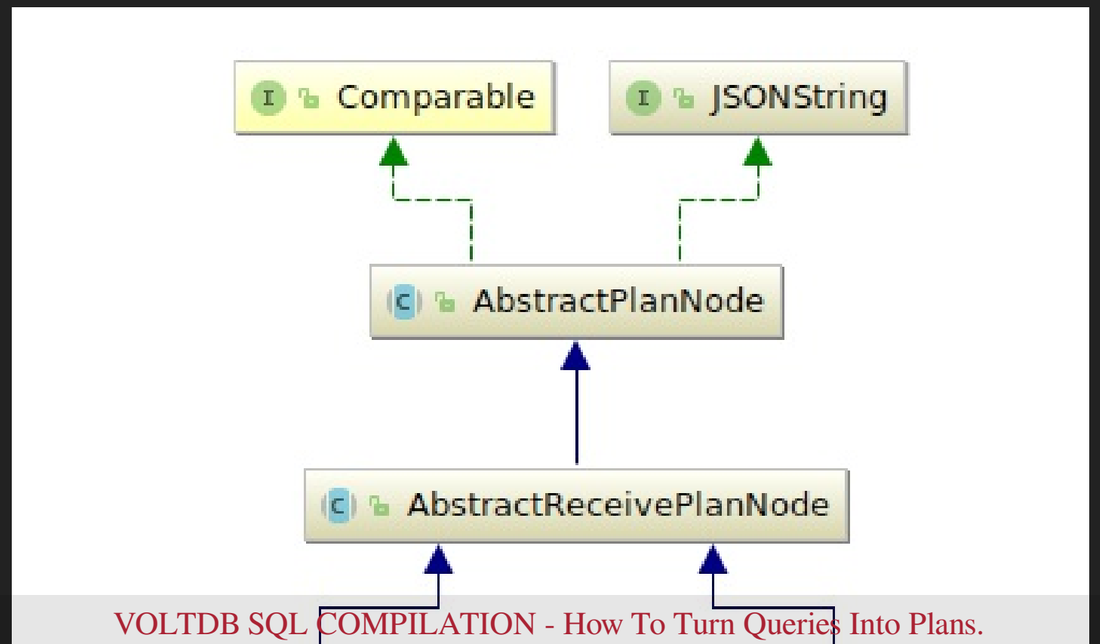
PlanNodes.1.a. Plan Node Details.

- Both Java and C++ have representations of plan nodes and whole plans, in JSON.
- There are about 24 different kinds of plan nodes.
- They are grouped into

Plans.1.a.i.
Operation Nodes.



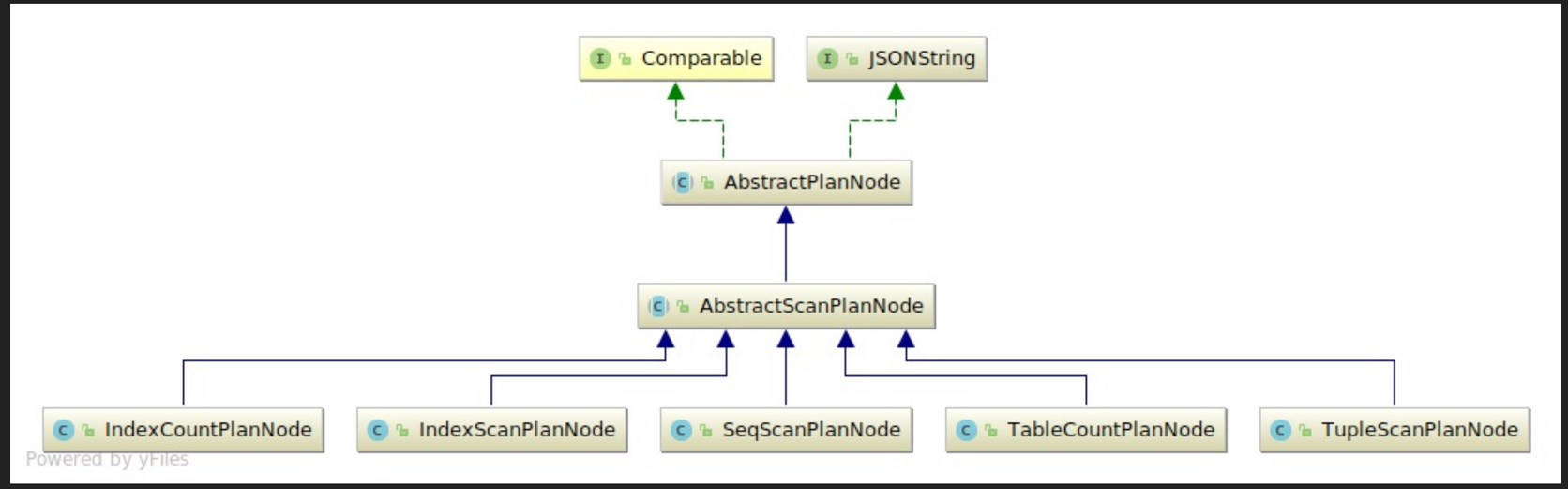
Plans.1.a.ii. ReceivePlanNodes



Plans.1.a.iii. Scan Plan Nodes.

- Sequential Scan nodes.
  - SEQSCAN nodes have a table name, and read all the rows in the table.
  - Filters which use only the table's columns can be put into the SEQSCAN node.
- Index Scan Nodes.

Plans.1.a.iv. Scan Node UML.

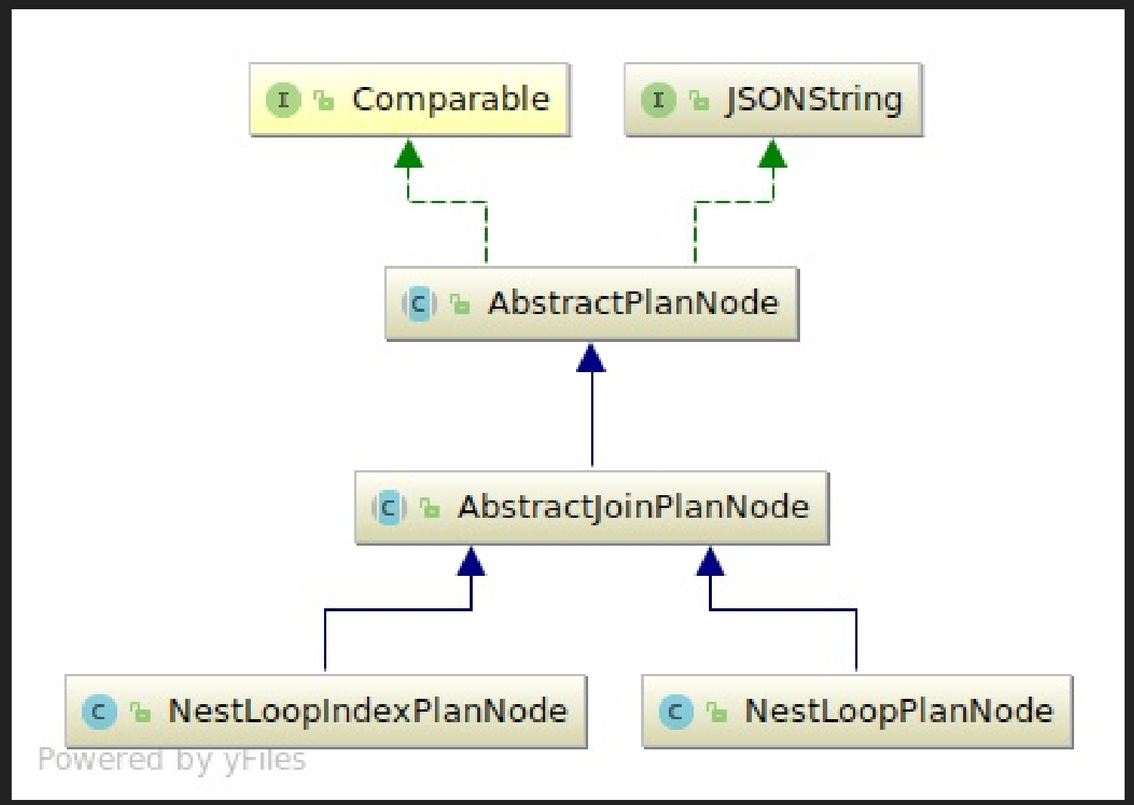


Plans.1.a.v. Join Plan Nodes.

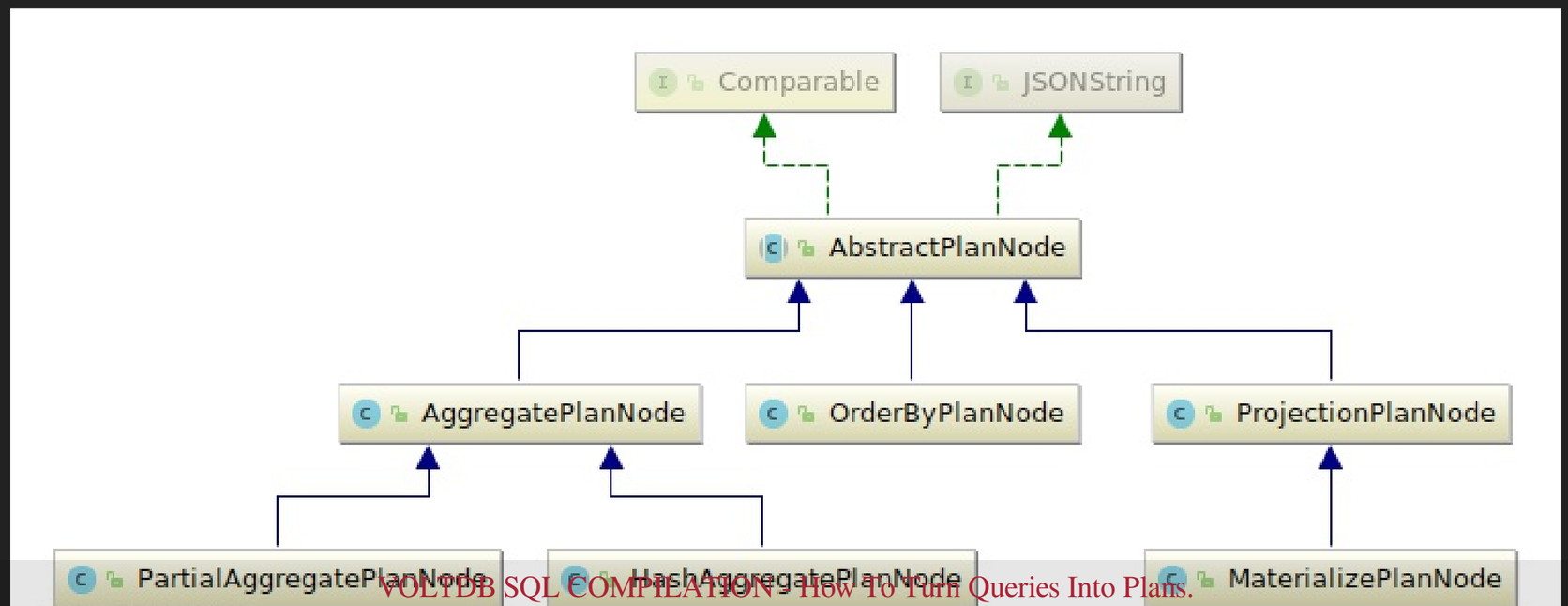
- There are a two kinds Join nodes, NESTLOOP and NESTLOOPINDEX.
- They both have exactly two children, the *outer* child on the left and the *inner* child on the right.

```
* Note: Don't confuse *outer* and *inner* here with outer
        and inner joins.
```

Plans.1.a.vi. Join Plan Nodes.



Plans.1.a.vii. Computation Plan Nodes.



VoltXMLElement
1. What is it?

VoltXMLElement objects are approximately what an XML parser might compile XML into.

- Each VoltXMLElement object has a name, like an XML element name.
- It also has a set of named, string valued attributes, like XML

VoltXMLElement
2. What is it good for?

We use VoltXMLElement objects whenever we need a handy, parsed representation of SQL.

- We use VoltXMLElement for a text based format for table and index

VoltXMLElement
2.a. An Example.

- select * from r join s on r.id = s.id;
- VoltXML:

```
<?xml version="1.0" encoding="UTF-8" ?>
<select>
  <columns>
    ...
  </columns>
  <parameters/>
  <tablescans>
    ...
  </tablescans>
```

VoltXML.2.b: Columns.

```
<columns>
  <columnref alias="ID"
             column="ID"
             id="1"
             index="0"
             table="R"/>
  <columnref alias="ID"
             column="ID"
             id="2"
             index="0"
             table="S"/>
</columns>
```

VoltXML.2.c: Scan of R

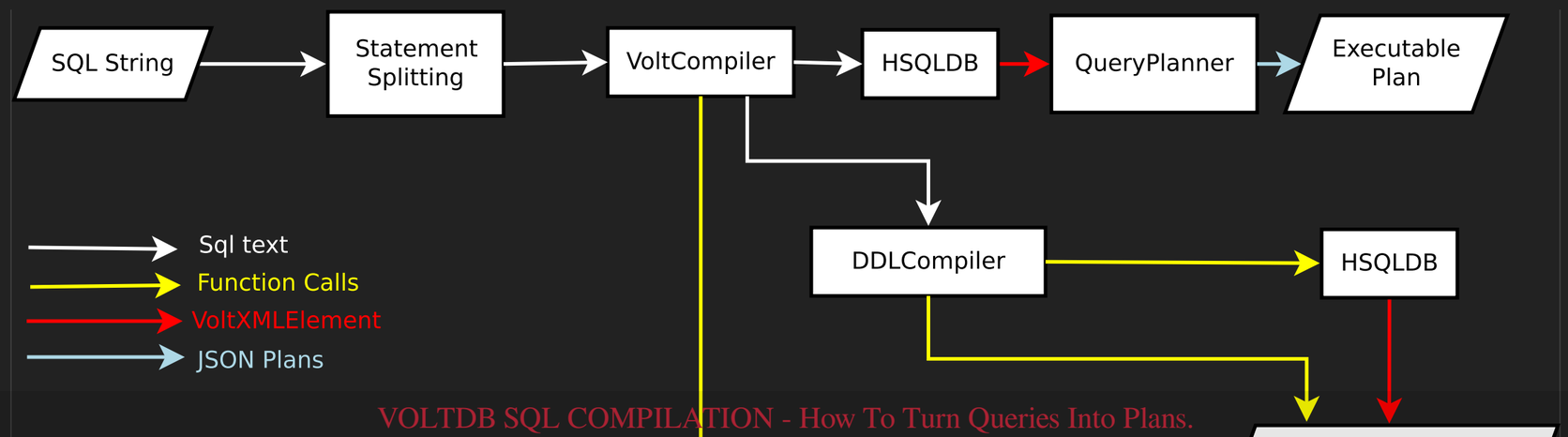
```
<tablescans>
  <tablescan jointype="inner"
             table="R"/>
  ...
</tablescans>
```

VoltXML.2.e: Scan of S.

```
<tablescans>
...
<tablescan jointype="inner" table="S">
  <joincond>
    <operation id="3"
               optype="equal">
      <columnref alias="ID" column="ID" id="2" index="0"
                 table="S"/>
      <columnref alias="ID" column="ID" id="1"
                 index="0"
                 table="R"/>
    </operation>
  </joincond>
</tablescan>
```

(Abstract) ParsedStatement s.1.

- Remember the flow diagram:



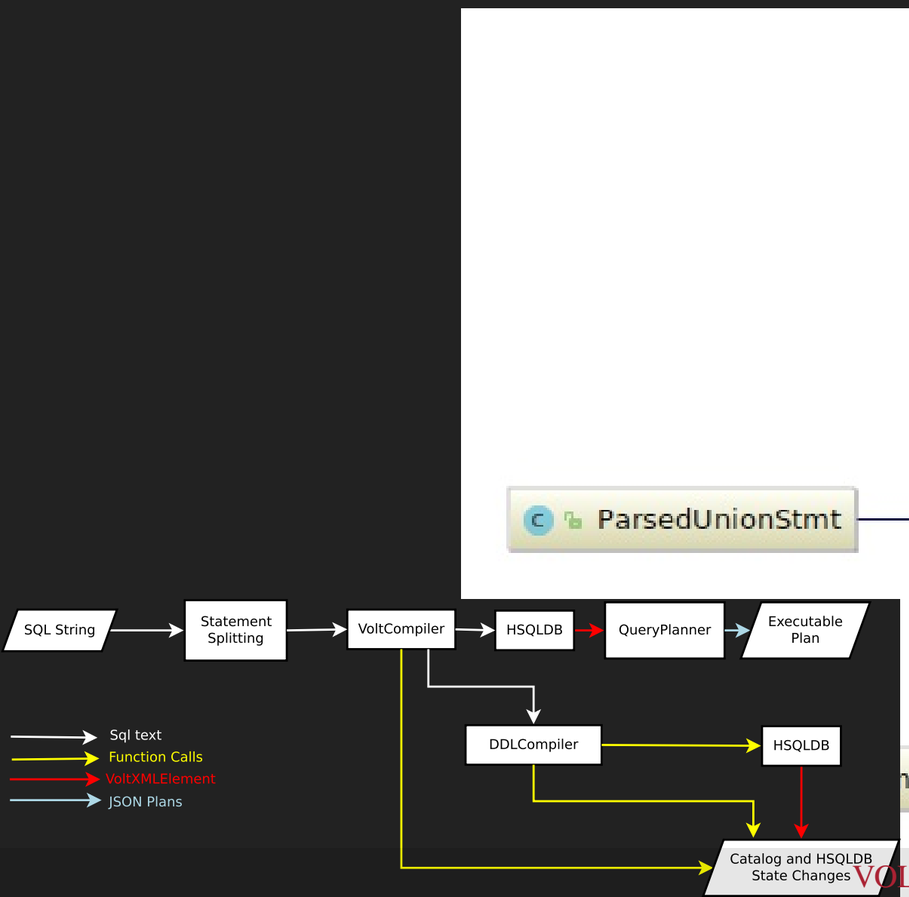
(Abstract) ParsedStatement s.2.

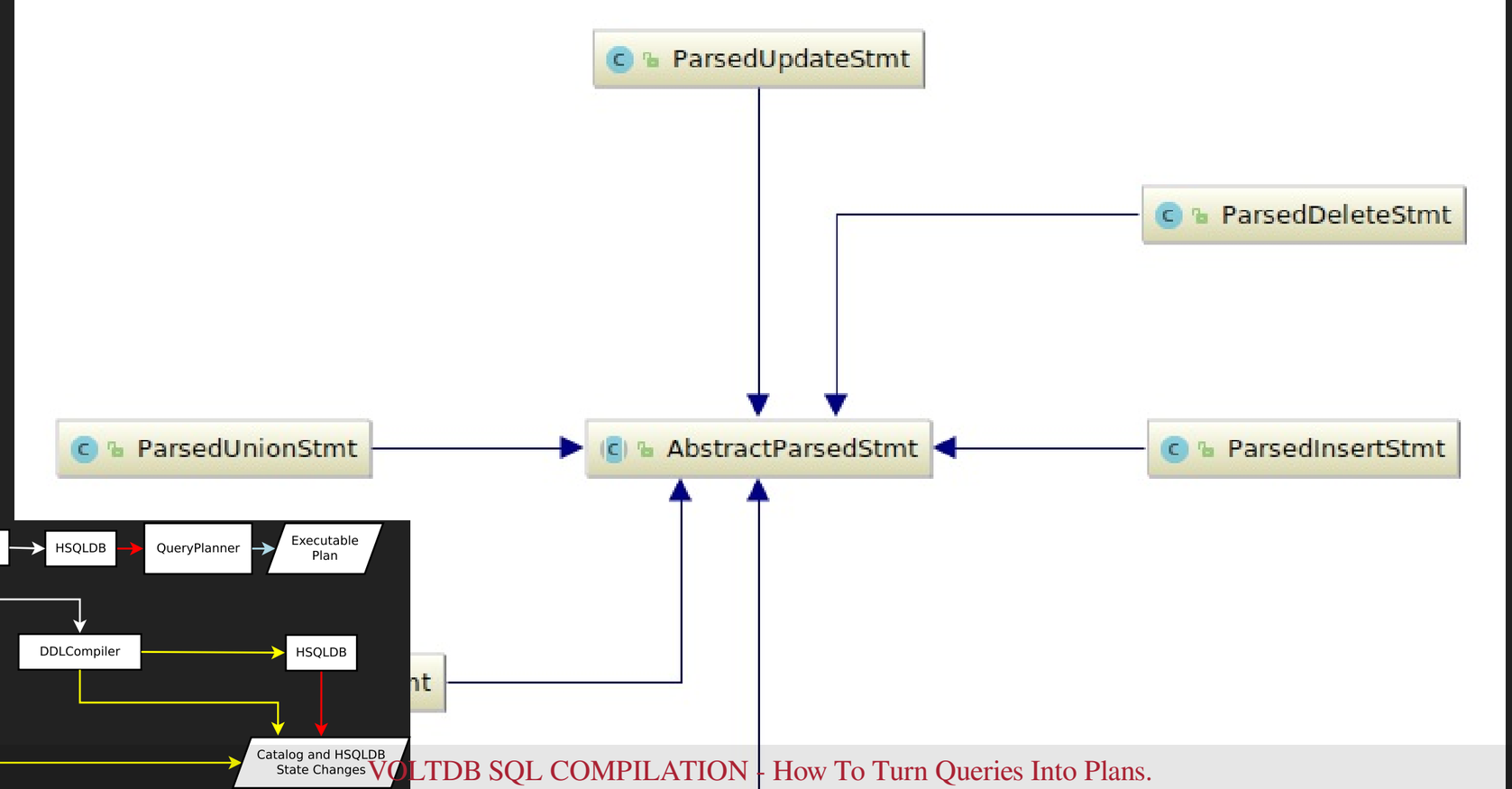
- An *AbstractParsedStatement* is VoltDB's internal representation of a SQL statement.

ment can be DQL (queries) or

sert/upsert/delete/swaptables).

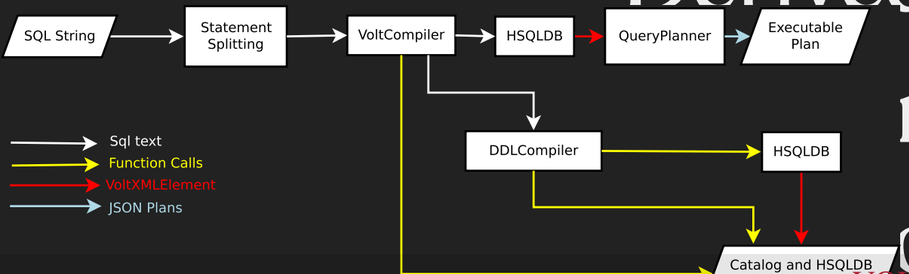
AS.1. UML Diagram.





AS.2. Embedded AbstractParsedStmts

- An AbstractParsedStmt may contain other AbstractParsedStmts.
  - Derived tables, which we call
    ries.
    on Tables, introduces with the



Planning. How the Planner Creates Plans.

- Three classes.
  - QueryPlanner
  - SubPlanAssembler
  - PlanAssembler

QueryPlanner

- The QueryPlanner.
  - Coordinates the SubPlanAssembler and the PlanAssembler

SubPlanAssembler

- The SubPlanAssembler.
  - Generates a set of join trees, with range variables in different orders.
  - Selects indexes where possible, forming AccessPaths.
  - Tries to rationalize index use for filters, window functions, order by and

PlanAssembler

- Gets subplans from the SubPlanAssembler.
- Adds nodes like aggregation, order by, window functions, receive/send pairs for multi partition plans.

QueryPlanner

- The QueryPlanner is called from the test system, via PlannerTestAideDeCamp, from the PlannerTool, and from the StatementCompiler.
  - PlannerTestAideDeCamp is a testing framework.
  - PlannerTool is the main planning tool, and is called from

QueryPlanner.a

- In QueryPlanner.parse, this calls the HSQLInterface to parse SQL text to form VoltXMLElement.
- In QueryPlanner.plan, calls the PlanAssembler.getBestCostPlan to generate the best plan.

'outputCompiled X' functions.

- These functions, outputCompiledPlan and outputCompiledStatement, are debugging functions.
- They write out intermediate forms of a SQL query.
- They may not do anything.

getBestCostPlan.

- In PlanAssembler.
- Generates plans for ephemeral scans.
- Generates plans for subquery (scalar) expressions.
- Generates plans for the main query.
- Repeatedly calls getNextPlan and compares it with the best plan.
- At this level we are dealing with entire plans.

getBestCostPlan.b

- Calculates determinism characteristics.
- Stitches together the plans for ephemeral scans and the main plan.

getNextPlan.a

- Delegate to a function for select, insert, delete, union, swap, update.
- The getNextSelectPlan is the most interesting.

getNextSelectPla

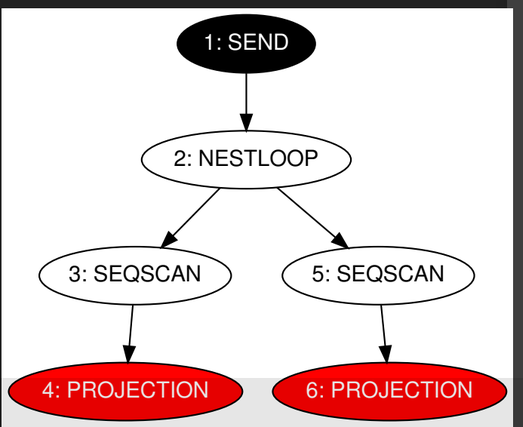
- Get the next subplan from the SubPlanAssembler.
- Add on an AGGREGATE_PLAN_NODE or HASH_AGGREGATE_PLAN_NODE if necessary.
- Add on a WINDOWFUNCTION_PLAN_NODE if necessary.

'select distinct' handling.

- 'select distinct' with no group by is converted to select with a group by the display list.
- 'select distinct' with a group by has the display list expressions.
- This complicates aggregation for Large Temp Tables.

Plans.1.d+. JSON formats

```
SQL: select * from r join s on r.id = s.id
COST: 6000000.0
PLAN:
{
    "EXECUTE_LIST": [
        3,
        5,
        2,
        1
    ],
    "IS_LARGE_QUERY": false,
    "PLAN_NODES": [
        {
            "CHILDREN_IDS": [2],
```



Common Table Planning.a.

- A Common Table is introduced with a WITH token.
- Example:

```
WITH RECURSIVE RT(ID, NAME) AS (
SELECT ID, NAME FROM R
  UNION ALL
SELECT ID, NAME FROM S WHERE RT.ID = S.ID
) SELECT ID, NAME FROM RT;
```

Common Table Planning.b.

- We may need to plan one or two statements.
  - If the query is not recursive, there is just one.
- We need to introduce the common table RT for the second SELECT statement, but not the first.

Catalogs

- A catalog has two forms.
  - A tree structure in memory.
  - A list of catalog commands as a disk file.
- Its structure is defined in voltdb/src/catgen/spec.txt.

voltdb/src/catgen/spec.txt format.

```
begin Table                          "A table (relation) in the databa
  // Comments are possible.
  Column* columns                      "The set of columns in the t
  Index* indexes                     "The set of indexes on the co
  Constraint* constraints              "The set of constraints on t
  bool isreplicated                  "Is the table replicated?"
  Column? partitioncolumn                "On which column is the
  int estimatedtuplecount              "A rough estimate of the nu
  ...
end
```

Catalog Commands.

- Stored in the jar file: voltdbroot/config/catalog.jar.
- Unjar this to get the file catalog.txt.
- Read and weep.

catalog.txt
Defining a table.

```
add /clusters#cluster/databases#database tables R
set /clusters#cluster/databases#database/tables#R isreplicated
set $PREV partitioncolumn null
set $PREV estimatedtuplecount 0
set $PREV materializer null
set $PREV signature "R|b"
set $PREV tuplelimit 2147483647
set $PREV isDRed false
```

catalog.txt defining a column

```
add /clusters#cluster/databases#database/tables#R columns ID
set /clusters#cluster/databases#database/tables#R/columns#ID
set $PREV type 6
set $PREV size 8
set $PREV nullable true
set $PREV name "ID"
set $PREV defaultvalue null
set $PREV defaulttype 0
set $PREV aggregatetype 0
set $PREV matviewsource null
set $PREV matview null
set $PREV inbytes false
```

MicroOptimizatio

- At a couple different times in planning we apply micro-optimizations.
- These are plan optimizations which are hard to do in the planner.
  - Typically the planner has only local information about a plan.
  - Sometimes information needed is not available when the plan is created.
- There are currently 9

MicroOptimization Strategy

- Implement apply, less common,
- Implement recursivelyApply, more common.

In either case there is a lot of hacking around with plan graph data structures.

Examples:

1. Remove identity projection nodes.
2. Inline aggregation.
3. Inline OrderBy into MergeReceive.
4. Inline insert.
5. Pushdown Limits.

Testing.

- We have two kinds of jUnit tests.
  - Planner tests, derived from PlannerTestCase.
  - Regression tests, derived from RegressionSuite.
- Planner tests.
  - They are fast.
  - Prefer using them if possible.
  - Debug them just like any Java

Planner Tests and plans.

Look at TestPlansLargeQueries for an example of a planner test.

```
validatePlan(
    "select max(aa) from r1 group by id, aa order by aa * aa",
    fragSpec(PlanNodeType.SEND,
            // project onto the display list.
            PlanNodeType.PROJECTION,
            // order by aa * aa
            PlanNodeType.ORDERBY,
            PlanNodeType.AGGREGATE,  // group by
            PlanNodeType.ORDERBY
```

Thank You for Your kind attention!