



**Hasso
Plattner
Institut**

IT Systems Engineering | Universität Potsdam

HYRISE Deep Dive

Martin Grund

Agenda

- Architecture
- Implementation
- Tasks

Status Quo

HYRISE is a prototype

- no transactions
- no guarantees
- plenty of fun!

Basic Features

- Vertical Partitioning
- Early / Late Materialization
- Compression

Architecture

- most important parts of HYRISE are a collection of shared libraries that provide database engine features exposed via an API
- infrastructure code for parallelization and remote access
- protocol is HTTP

Basic Modules

- **IO** - Loading data from disc, catalog management
- **Storage** - Table, data types, compression, met data
- **Access** - Query operators, predicates
- **Net** - communication
- **Helper** - Memory management, logging, tracing, ...

Installation

- User Guide (HPI network) – <http://goo.gl/47qc8>
- Ubuntu fast install – <http://goo.gl/I2Msh>

Installation

```
$ cp settings.mk.default settings.mk
```

```
$ make
touch settings.mk
make --no-print-directory -s --directory=build/jsoncpp
CC third_party/jsoncpp/json_writer.cpp
LINK build/libjson.dylib
ld: warning: directory not found for option '-L/usr/lib64'
make --no-print-directory -s --directory=build/lib/helper
DEPEND trace.d
DEPEND Settings.d
DEPEND RetainCountingObject.d
DEPEND prefetching.d
DEPEND logger.d
DEPEND HttpHelper.d
DEPEND hash.d
DEPEND GroupValue.d
DEPEND demangle.d
DEPEND ConfigFile.d
```


Developing With HYRISE

Where to Start?

- All (most) features are **tested**
 - `src/bin/units_access`
 - `src/bin/units_storage`

HYRISE Data Format

- CSV data + Header
- Header consists out of 4 rows
 - Name
 - Data Type
 - Grouping
 - Separator

HYRISE Data Format

```
int|float|string  
INTEGER | FLOAT | STRING  
0_R | 1_R | 1_R  
===  
3|3.1|null  
0|0.1|zwei  
2|2.1|vier  
5|5.1|doppelt  
7|7.1|acht
```

HYRISE Data Format

- Grouping defines vertical partitioning
- 0_R defines an identifier (0) and group type R (currently only R)
- Group identifiers must increase
- $0_R \mid 1_R \mid 1_R$ defines a two container table with three attributes

Loading Data

- **Loader** - construction/combination of loaders

```
namespace Loader
class params {
private:
#include "parameters.inc"
    param_ref_member (AbstractInput, Input);
    param_ref_member (AbstractHeader, Header);
    param_member (AbstractTableFactory*, Factory);
    param_member (string, BasePath);
    param_member (bool, InsertOnly);
public:
    params ();
    ~params ();
    params* clone () const;
};
AbstractTable* load (const params& args);
```

Loader Shortcuts

- Provides predefined options for loading data, less flexible, less code

```
namespace Loader {
    namespace shortcuts {
        /*!
            Loads a table for a given filename,
            assuming it contains header and data in HYRISE format.
            \param filepath
        */
        AbstractTable* load(const string& filepath);
        /*!
            Load a table from a given filename,
            while supplying the header from another file.
            Must be HYRISE formatted.
            \param datafilepath
            \param headerfilepath
        */
        AbstractTable* loadWithHeader(const string& datafilepath,
        const string& headerfilepath);
    };
};
```

Storage Manager

- Provides catalog management for HYRISE
(`src/lib/io/StorageManager.h`)
 - Encapsulates name, table mapping
 - Primarily used in plan operations
- Singleton class
- Support for binary dumps

Querying Data

The **easy** way:

```
SELECT a,b FROM table WHERE c = 9;
```

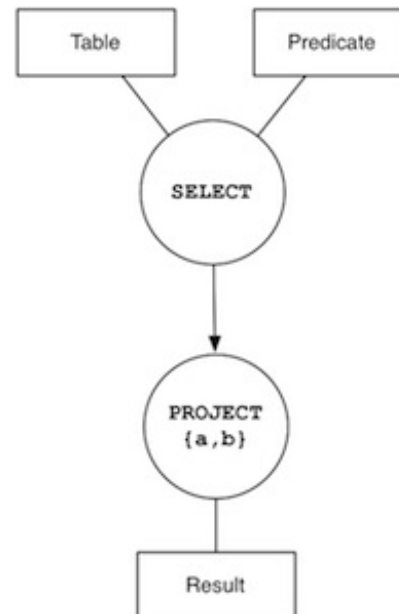
However, there is no easy way in HYRISE.

Querying Data

The original query

```
SELECT a,b FROM table WHERE c = 9;
```

becomes



Querying Data

```
SELECT a,b FROM table WHERE c = 9;
```

becomes

```
AbstractTable *table =  
  Loader::shortcuts::loadWithHeader("data",  
                                     "header");  
  
EqualsExpression<int>* eq =  
  new EqualsExpression<int>(table,  
                             table->numberOfColumn("c"), 9);  
  
SimpleTableScan stc;  
stc.setPredicate(eq);  
stc.addInput(district);  
stc.setProducesPositions(true);  
AbstractTable *positions = stc.execute();  
  
ProjectionScan ps;  
ps.addInput(positions);  
ps.addField(table->numberOfColumn("a"));  
ps.addField(table->numberOfColumn("b"));  
AbstractTable *result = ps.execute();
```

HYRISE Remote Access (a.k.a server)

- Build on the ideas of others
 - HTTP as database protocol
 - Queries as post requests with valid JSON
- Allows to use your favorite language to query HYRISE using HTTP

How things get done

- Smallest execution unit is a task – plan operations derive from tasks
- Task are ordered by their dependencies
- Tasks can form any cycle-free graph

Client - Bad Request

```
$ curl http://localhost:5000
```

Client - Bad Request

```
$ curl http://localhost:5000 -d "query=`cat test`"
```

Server - Answer

```
$ ./build/hyrise_server 5000  
Warning: no body received!  
127.0.0.1 [2011-11-10 11:08:31 +0100] GET / (0.0
```


Client - Good Request

```
curl http://localhost:5000 -d "query=`cat test/autojson/howto.json`"

{"header" : [ "col_0", "col_1" ],
"message_handling" :
{
  "parse_request" : 1999,
  "respond_request" : 116
},
"performanceData" :
[
  {
    "data" : 0,
    "duration" : 2415,
    "name" : "TableLoad",
    "papi_event" : "PAPI_TOT_INS"
  },
  {
    "data" : 0,
    "duration" : 26,
    "name" : "ProjectionScan",
    "papi_event" : "PAPI_TOT_INS"
  }
],
"rows" :
[
  [ 0, 1 ],
  [ 200, 300 ],
  [ 400, 500 ],
```

Server - Answer

```
$ HYRISE_DB_PATH=`pwd`/test ./build/hyrise_server
2011-11-10 11:29:20,136 [0x102281000] INFO hyrise.net - Json received
{
  "edges" :
  [
    [ "0", "1" ]
  ],
  "operators" :
  "-1" :
  {
    "filename" : "lin_xxxs.tbl",
    "table" : "reference",
    "type" : "TableLoad"
  },
  "0" :
  {
    "filename" : "lin_xxxs.tbl",
    "table" : "lin_xxs",
    "type" : "TableLoad"
  },
  "1" :
  {
    "fields" : [ 0, 1 ],
    "type" : "ProjectionScan"
  }
} } }
2011-11-10 11:29:20,137 [0x102304000] DEBUG access.plan.PlanOperat
2011-11-10 11:29:20,138 [0x102387000] DEBUG access.plan.PlanOperat
```

Behind The Curtains

Raw Data Access

- Every data container in HYRISE implements the *AbstractTable* interface * only 1 container actually stores data

```
class Table{ }
```

- `Table` is the base container type, no further decomposition

Raw Data Access - Type Handling

- Type handling is deferred to C++ templates
- However, template methods cannot be overloaded
 - Compression helps
 - Typical operations work with value ids

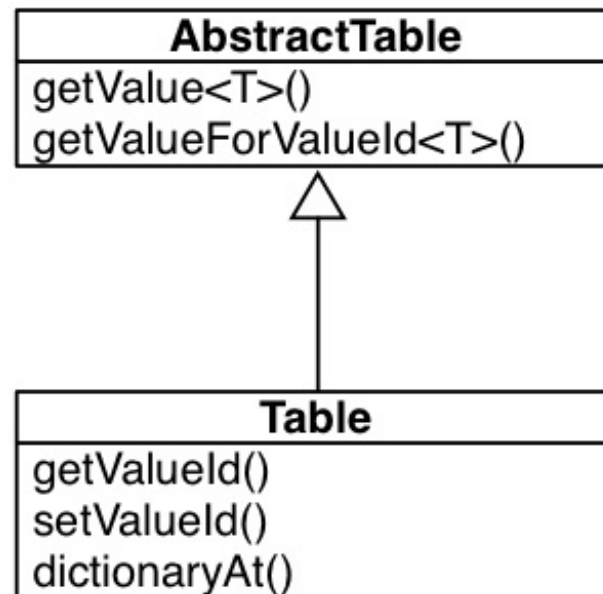


Table Composition

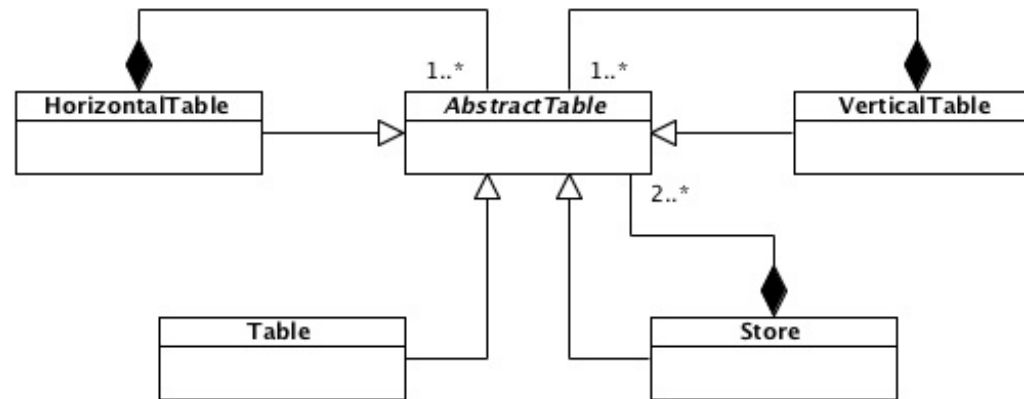


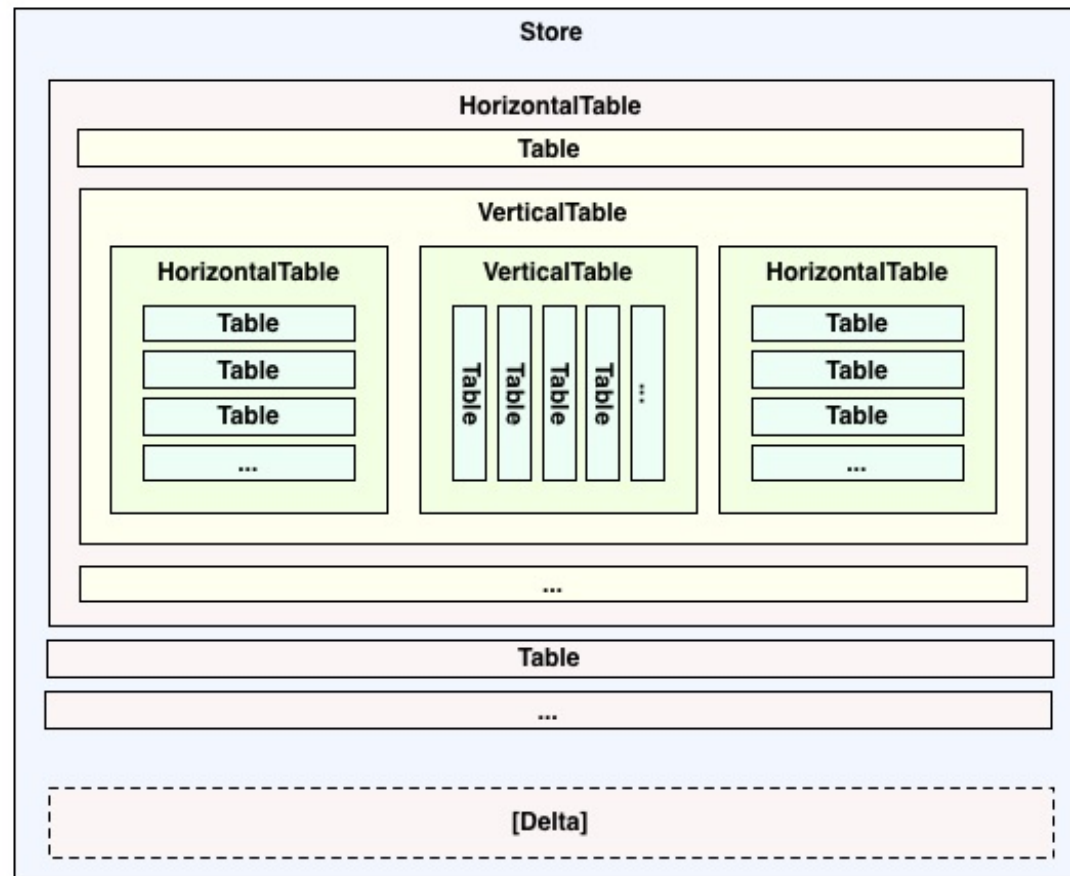
Table Composition

- **AbstractTable** is the magic base class for all data storages, it is used in many different ways, in containers, plain tables, intermediate results and more.
- **Table** is the innermost entity in the table structure. It stores the actual values like a regular table and cannot be split further.
- **VerticalTable** implements a vertical table layout. It is organized into one or more vertical entities spreading over a distinct number of columns. The entities can be any subclass of AbstractTable.

Table Composition (II)

- **HorizontalTable** implements a horizontal table layout. It is organized into one or more horizontal subtables spreading over a distinct number of rows. The subtables can be any subclass of AbstractTable.
- **Store** consists of one or more main tables and a delta store and is the only entity capable of modifying the content of the table(s) after initialization via the delta store. It can be merged into the main tables using a to-be-set merger which defines the merge strategy.

Table Composition



Compression

- Dictionary compression is always enabled, immutable tied to the storage system
- Infrastructure makes it easy to plug other compression algorithms easily

Memory Management

- C++ provides no easy way out (compared to C++11)
- Manual retain counting (thread-safe,
`src/lib/helper/RetainCountingObject.h`)

Memory Management

```
struct Test
{
    AbstractTable* table;

    void setTable(AbstractTable* t)
    {
        table = t;
        table->retain();
    }

    ~Test()
    {
        table->release();
    }
};
```

New Plan Operators

- Operators can perform any action, as long as they implement `_PlanOperation`
- Use existing unit tests and other plan operators as template

```
class _PlanOperation : public OutputTask
{
protected:
    virtual void setupPlanOperation();
    virtual void executePlanOperation() = 0;
    virtual void teardownPlanOperation();

public:
    void setLimit(unsigned long l);
    void setProducesPositions(bool p);
    void setFields(field_list_t *fields);
    void addField(field_t field);
    void addInput(AbstractTable *input);
    void addInput(vector<AbstractTable *>* input_list);
}

// Implement own operation
class MyOperation : public _PlanOperation
{
    void executePlanOperation()
    {
        /* ... */
        addResult(myTable);
    }
}
```