The Pyrrho Experiment

MALCOLM CROWE AND FRITZ LAUX

DBKDA 2022
Malcolm Crowe is an Emeritus Professor at the University of the West of Scotland, where he worked from 1972 (when it was Paisley College of Technology) until 2018.

He gained a D.Phil. in Mathematics at the University of Oxford in 1979.

He was appointed head of the Department of Computing in 1985. His funded research projects before 2001 were on Programming Languages and Cooperative Work.

Since 2001 he has worked steadily on PyrrhoDBMS to explore optimistic technologies for relational databases and this work led to involvement in DBTech, and a series of papers and other contributions at IARIA conferences with Fritz Laux, Martti Laiho, and others.

Prof. Crowe has recently been appointed an IARIA Fellow.
Prof. Dr. Fritz Laux was professor (now emeritus) for Database and Information Systems at Reutlingen University from 1986 - 2015. He holds an MSc (Diplom) and PhD (Dr. rer. nat.) in Mathematics.

His current research interests include

- Information modeling and data integration
- Transaction management and optimistic concurrency control
- Business intelligence and knowledge discovery

He contributed papers to DBKDA and PATTERNS conferences that received DBKDA 2009 and DBKDA 2010 Best Paper Awards. He is a panellist, keynote speaker, and member of the DBKDA advisory board.

Prof. Laux is a founding member of DBTech.net (http://www.dbtechnet.org/), an initiative of European universities and IT-companies to set up a transnational collaboration scheme for Database teaching. Together with colleagues from 5 European countries he has conducted projects supported by the European Union on state-of-the-art database teaching.

He is a member of the ACM and the German Computer Society (Gesellschaft für Informatik).
This presentation

- The result of the Pyrrho v7 experiment
  - To reimplement Pyrrho to use
    - Shareable data structures throughout
    - Without sacrificing any features

- Background
- Ground Rules
- Progress since DBKDA 2021 Tutorial
  - Highlights and insights
- Future steps

Next: Background
Background

DKBDA 2018 TO NOW

Next: DBKDA 2018
DBKDA 2018-now

- A sequence of contributions
  - Transaction focus, optimistic execution
  - Looking at Big Live Data
  - Always prioritising correctness over speed

- We demonstrated StrongDBMS
  - at DBKDA 2019 with code examples
  - A tiny optimistic DBMS, *serialized* log file
  - Outperformed all other relational DBMS
  - In a test featuring high concurrency

- To show that optimistic can be best

- Next: “Serializable” Tx
“Serializable” Transactions

- The goal of any DBMS should be to serialise transactions
  - the computer scientist wants that
- It can be done, but it takes time
  - and brings limitations (2 army problem)
- But most DBMS customers want speed
  - and say correctness is less important 😞
- So trade-offs are inevitable

Next: Productivity vs ..
Productivity vs Safety

- Commercial DBMS (Oracle, SQL Server etc)
  - recommend avoiding serializable requirement
  - In 2019 we showed that requiring serialisability
    - made many transactions fail in a TPCC demo
    - throughput fell off at about 30 clerks
- In the same demo we had StrongDBMS prototype
  - Guaranteed transactions serialised
  - Outperformed other DBMS in productivity
    - throughput continued to increase beyond 100 clerks
  - But was a very simple DBMS
    - lacking many features DB people expect
  - With a radical approach to data structures

- Shareable Data Structures
Shareable Data Structures

- StrongDB’s magic ingredient was
  - SHAREABLE data structures throughout
- Our favourite optimistic DBMS Pyrrho
  - Did not perform well in the test
- It was natural to re-engineer Pyrrho
  - To use shareable data structures
- So that it would be equally good
  - In high concurrency situations

Next: Benefit of Shareable
What we gain from Shareable

- Build in the notion of transaction isolation
  - at all levels of implementation
- Structures are shared but never copied
  - Immutable, all fields readonly or final
- A changed object has a new root node
  - Shares all the old ones with previous version
- Brings great advantages for transactions
  - Isolation, instant snapshot, just forget on rollback
  - But is more complex to program

Next: When we add a node
When we add a node
Transaction and B-Tree

M. K. Crowe, S Matalonga: StrongDBMS: Built from Immutable Components
StrongDBMS vs PyrrhoDBMS

- StrongDBMS had simple tables
  - No triggers, alter/drop, procedures ..
  - SQL parsing done on the client
- (Just enough capability for TPCC)
- Both DBMS have persistent tx log
  - Serialized is stronger than serializable
  - and optimistic transaction execution

Next: Pyrrho fared poorly
Pyrrho DBMS fared poorly

- It is optimistic and has serialized tx log
- But also obsessive, too many features
  - Triggers, Cascades, User Defined Types
  - Object oriented database objects
  - Etc etc
- We found that this ambition is too much
  - Safe but not good for high concurrency
  - Outperformed by all other RDBMS
- But – it has RESTViews, big live data...

Next: Big Live data..
Big Data and Big Live Data

The problems with Big Data
- Data is dead, always out of date
- Correct only at the time it was extracted
- Taken out of context, not evolving

With Big Live Data, data is accessible
- From the source where it lives, evolves

View-mediated data warehousing
- Using REST for integration
- PyrrhoDB does this really well

Next: The ground rules
The ground rules
Everything must be shareable

- All fields are readonly and shareable
  - Can only be given values in constructor
- Might lead to very long argument lists
  - Unless we use idea of a property tree
  - Have += operator to add a property value
- Pyrrho v7 does this
  - Subclasses provide such tree to the base()
  - Relocation cascades changes to fields
  - Cascades for replacement of an object

Next: Mechanisms
Tree structures

- BTree<K,V> is immutable, shareable
  - When K and V are shareable
  - Two-way traversal uses immutable bookmarks

- Database, Transaction all shareable
  - Contents (tables etc) must be shareable too

- Transaction is a private copy
  - Increments are prepared and committed

- Database is built from the tx log

Next: Shareable DB objects
Shareable database objects

- From SQL syntax
  - Table, Domain, Column, View, Role etc
  - SQL expressions, literals, functions etc
  - SQL statements for DML and stored modules

- TypedValue classes with domain
  - TInt, TChar, TBlob, TRow, TArray etc

- RowSets for collecting results
  - RowSets form a pipeline from base tables
  - Some are updateable
  - Cursors are a kind of bookmark of TRow

Next: The first steps
The first steps in the experiment

- In 2019 it was safe but not productive
  - As many other DBMS were (e.g. PostGRES)
  - Both became unproductive above 6 clerks
- Would shareable data structures help?
- In 2021 a **V7 demo** with good productivity
  - Productivity increasing up to 50 clerks
- But lacked many advanced DBMS features
  - And did not have quite the right structures
- The V7alpha experiment continued

Next: Objectives of the experiment
Objectives of the experiment

- Pyrrho with shareable data structures
  - Can it be done for all features?
    - Even RESTView? Optimistic execution?
  - Would it fix the transaction performance?
  - What programming lessons can be learned?
  - Would it become unusably slow?
    - (StrongDBMS was fast and had shareable d.s.)

Next: Since DBKDA 2021
Since DBKDA 2021
Progress since DBKDA 2021

- What happens to PyrrhoDBMS
  - .. if we require shareable data structures?
  - The extra complexity slows performance
to unproductive levels

- TPCC has realistic requirements
  - 1 clerk can only enter 16 orders in 10 mins

- In 2021 Pyrrho still could almost do this
  - But with 2022 version of Pyrrho only 10

- This may improve with further development

Next: TPCC new order
A New Order in progress

**TPC/C**

**Setup | New Order | Order Status | Payment**

**New Order**

- **Warehouse:** 1
- **District:** B
- **Customer:** 53
- **Name:** BARESEPRI
- **Credit:** GC
- **%Disc:** .3940
- **Order Number:** 3003
- **Number of Lines:** 12
- **W_tax:** .1368
- **D_tax:** .1272

<table>
<thead>
<tr>
<th>Supp_W</th>
<th>Item_Id</th>
<th>Item Name</th>
<th>Qty</th>
<th>Stock</th>
<th>B/G</th>
<th>Price</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>79744</td>
<td>TTRCA EMQTBH KAKGGK</td>
<td>7</td>
<td>51</td>
<td>G</td>
<td>$16.25</td>
<td>$113.75</td>
</tr>
</tbody>
</table>

**Execution Status:**

**Total:**

Next: Parsing and Queries
Parsing and query analysis

- Analyse SQL from left to right
- Add known properties as we find them
- Create RowSets as soon as possible
- Adjust properties later via cascades
- SQL is complex
  - Pyrrho has become safer but slower
- Still unwilling to sacrifice correctness

Next: Rowsets replace Queries
RowSets replace Queries

- RowSets are immutable (of course)
  - They naturally form a tree by source
- The SQL standard: derived tables
- Instead of “optimising queries”
  - Think of the properties of RowSets
  - E.g. apply a where-condition, grouping
  - Change propagates to sources
- RowSet keeps track of suitable indexes
- And many RowSets are updatable

Next: Update a join
Example: update a join

- Many views and joins can be updated
  - e.g. if some of the columns are keys
  - in one or more of the joined base tables

- An update to the join then becomes
  - an update to one or more of the these tables
  - as table instances

- If the table is remote, we can use POST
SQL code parsed once only

- On definition of a view or procedure
- Then has its own unique identifiers
- Avoids conflict with similar names
- Similarly, table and view references
  - Instanced: new ids for their columns etc
- Uids are 64-bit longs, unique in the DB
- Each range of uids has size $2^{60}$

Next: Uids instead of names
Uid instead of identifier chains

- SQL identifiers get replaced by uids
- Unique Identifiers are just long integers
- Unique within the database/transaction
- Refer to a shareable database object
  - Column, Expression, Table, RowSet, Procedure,
- Committed objects uids are file location
- Others are private to the transaction
  - Can be lexical position in source SQL
  - Or ids of precompiled objects (view, proc)
  - Or allocated on a heap

Next: Virtual data warehousing
View-mediated REST access

▶ A view into live data (no copying)

[CREATE VIEW sales_V
   (customer, sales, accSalesShare)
AS SELECT customer, sales,
   (SELECT SUM(sales) FROM custSale
    WHERE sales >= u.sales) /
   (SELECT SUM(sales) FROM custSale)
FROM custSale AS u]

▶ Designed for filtering by item
  ▶ To discourage retrieval of the entire table

Next: SQL for ABC-Analysis
Example: ABC-Analysis

- Originally, ABC-analysis is a clustering of customers with regard to their contribution to the sales of a company
  - A-customers contribute the most, B is medium, and C-group customers are least
  - The algorithm is defined by 2 threshold values (t1, t2) which separate A from B and B from C group
  - These values are usually t1=50% and t2=85%

<table>
<thead>
<tr>
<th>ABC</th>
<th>Customer</th>
<th>Sales</th>
<th>accum.S.</th>
<th>...</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Daimler</td>
<td>20000</td>
<td>20000</td>
<td></td>
<td>26%</td>
</tr>
<tr>
<td>A</td>
<td>Bosch</td>
<td>17000</td>
<td>37000</td>
<td></td>
<td>48%</td>
</tr>
<tr>
<td>B</td>
<td>Siemens</td>
<td>10000</td>
<td>47000</td>
<td></td>
<td>60%</td>
</tr>
<tr>
<td>B</td>
<td>Stihl</td>
<td>8000</td>
<td>55000</td>
<td></td>
<td>71%</td>
</tr>
<tr>
<td>B</td>
<td>VW</td>
<td>7000</td>
<td>62000</td>
<td></td>
<td>80%</td>
</tr>
<tr>
<td>C</td>
<td>Porsche</td>
<td>5000</td>
<td>67000</td>
<td></td>
<td>86%</td>
</tr>
<tr>
<td>C</td>
<td>Migros</td>
<td>4000</td>
<td>71000</td>
<td></td>
<td>91%</td>
</tr>
<tr>
<td>C</td>
<td>ATU</td>
<td>3500</td>
<td>74500</td>
<td></td>
<td>96%</td>
</tr>
<tr>
<td>C</td>
<td>Ford</td>
<td>3200</td>
<td>77700</td>
<td></td>
<td>100%</td>
</tr>
</tbody>
</table>

Next: SQL for ABC
Let table custSale (customer, sales)

Query sales_V and assign a group to each customer according to its sales percentage ordered by descending sales values.

```
[SELECT CASE
     WHEN accSalesShare <= 0.5 THEN 'A'
     WHEN accSalesShare > 0.5 AND accSalesShare <= 0.85 THEN 'B'
     WHEN accSalesShare > 0.85 THEN 'C'
     ELSE NULL
     END as ABC,
     customer, sales,
     CAST(CAST(sales/ (SELECT SUM(sales) FROM sales_V) * 100 as decimal(6,2)) as char(6)) || ' %' AS share
     FROM sales_V
     ORDER BY sales DESC]
```

Result

```
<table>
<thead>
<tr>
<th>ABC</th>
<th>CUSTOMER</th>
<th>SALES</th>
<th>SHARE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Daimler</td>
<td>20000.00</td>
<td>25.74 %</td>
</tr>
<tr>
<td>A</td>
<td>Bosch</td>
<td>17000.00</td>
<td>21.88 %</td>
</tr>
<tr>
<td>B</td>
<td>Siemens</td>
<td>10000.00</td>
<td>12.87 %</td>
</tr>
<tr>
<td>B</td>
<td>Stihl</td>
<td>8000.00</td>
<td>10.30 %</td>
</tr>
<tr>
<td>B</td>
<td>VW</td>
<td>7000.00</td>
<td>9.01 %</td>
</tr>
<tr>
<td>C</td>
<td>Porsche</td>
<td>5000.00</td>
<td>6.44 %</td>
</tr>
<tr>
<td>C</td>
<td>Migros</td>
<td>4000.00</td>
<td>5.15 %</td>
</tr>
<tr>
<td>C</td>
<td>ATU</td>
<td>3500.00</td>
<td>4.50 %</td>
</tr>
<tr>
<td>C</td>
<td>Ford</td>
<td>3200.00</td>
<td>4.12 %</td>
</tr>
</tbody>
</table>
```

Next: No query rewriting
No query rewriting

- Consider the <select list> concept in View
- If it contain aggregation functions
  - AVG, MAX, MIN, SUM, EVERY, ANY, COUNT, STDEV.., COLLECT, FUSION, INTERSECTION
- During rowset traversal rows get added in:
  - The resulting rowset has one row per group
  - Rows in the source are added in to the result rowset
  - Using Registers containing various accumulators, sums, multisets, ..
- Now suppose the view is remote (use REST)
  - Sending it to a list of remote contributors
- This used to require a lot of analysis and rewriting extra column names for the remote query
- COUNT becomes SUM, AVG needs SUM and COUNT, STDEV needs sums of squares, collections..
- We don’t need to do this any more

Next: How REST works
What happens with REST

- REST operations use standard formats
- For rows we use JSON documents
- An item for each column of the row
- Why not add some extra columns for the Registers in that row?
- There is a Register for each occurrence of an aggregation function in the select list
- We define how to represent a Register in JSON

Next: an example
A RESTView example

- With several remote sources via POST
- Grouped aggregations are interesting

```sql
select sum(e)+char_length(f), f from ww
  group by f
```

- We no longer rewrite it, but send as is:

```ruby
http://localhost:8180/DB/DB select (SUM(E)+CHAR_LENGTH(F)), F from t group by F
Returning ETag: "23,-1,180"
--> 4 rows
Response ETag: 23,-1,180
http://localhost:8180/DC/DC select (SUM(E)+CHAR_LENGTH(F)), F from u group by F
Returning ETag: "23,-1,159"
--> 3 rows
Response ETag: 23,-1,159
```
How does this work?

- Each database returns its answer
- The data from each has extra fields
- The Registers for aggregates by group
- Unpacked and combined by Pyrrho

Next: The extra fields
Extra Register fields

- The local and remote servers see the same value expression
  - So the registers are supplied in the left-to-right ordering

- As a Json document with the following items:
  - The string value accumulated by the function if any
  - The value of MAX, MIN, FIRST, LAST, ARRAY
  - A document containing numbered fields for a multiset value
  - The value of a typed SUM
  - The value of COUNT
  - The sum of squares (if required for standard deviation etc)

Next: Transactions and REST
Transactions and REST

- Because of the two-army problem
  - At most one remote participant
- A set of commit steps is agreed
- The local DB starts the commit
- If the remote DB reports success
- The local DB can complete the commit

Next: The Result of the Experiment
The result of the experiment

- Pyrrho v7 uses shareability throughout
  - Safe in high concurrency situations
  - It implements Big Live Data protocols
  - But it is slower
- It showcases optimistic execution
- And in some ways is a model to follow

Next: Future Steps
Future steps

Next: for Pyrrho DBMS
Next steps for PyrrhoDBMS

- From alpha to beta..
- Versioned object Web applications API
  - Based on POCO (plain old C# objects)
- US DoD “Orange book” security
- Some support for Java
- Finish Window functions

Next: Working with other DBMS
Working with other DBMS

- REST for server communication
  - Common format (JSON), protocol (HTTP1.1)
  - Possibly with ETags (RFC7232), Registers
- As a non-privileged Internet client
  - With privileges allocated in the usual way
- Need adaptation to SQL dialects
- Agreement about transactions
  - Avoid two-army problem

Next: References
Links

Crowe, M. K., Matalonga, S.: Shareable Data Structures, on https://github.com/MalcolmCrowe/ShareableDataStructures

- includes source code for StrongDBMS, PyrrhoV7alpha and documentation

Crowe, M. K., Laux, F.: Implementing True Serializable Transactions, Tutorial, DBKDA 2021

- https://www.youtube.com/watch?v=t4h-zPBPtSw&t=39s
- https://www.iaria.org/conferences2021/filesDBKDA21/

- Version 6.3: https://pyrrhodb.uws.ac.uk
- 50 clerks demo: https://youtu.be/0YaU59LvgLs
- Pyrrho blog: https://pyrrhodb.blogspot.com

Next: References
References


Crowe, M. K., Matalonga, S., Laiho, M: StrongDBMS, built from immutable components, DBKDA 2019

Crowe, M. K., Fyffe, C: Benchmarking StrongDBMS, Keynote speech, DBKDA 2019

Crowe, M. K., Laux, F.: DBMS Support for Big Live Data, DBKDA 2018

Crowe, M.K., Begg, C.E., Laux, F., Laiho, M: Data Validation for Big Live Data, DBKDA 2017