An Optimistic, RESTful, Serialized Relational Database Management System using immutable structures

MALCOLM CROWE AND FRITZ LAUX

IARA CONGRESS 2022

Malcolm Crowe

University of the West of Scotland
Email: malcolm.crowe@uws.ac.uk

- 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

(Retired), Reutlingen University
Email: fritz.laux@reutlingen-university.de

- 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

- Relational Database Management System (DBMS)
- Pyrrho DBMS is a research prototype whose development began in 2005
  - Optimistic Execution
  - Serialized Transactions
- Shareable Data Structures
- Big Live Data
  - Virtual Data Warehousing
- A Versioned RESTful library for web apps
  - Application Programming Interface (API)
- Conclusions and future steps

- Next: Serialized transactions

Serialized Transactions

- The goal of any DBMS
  - Should be to serialize transactions
  - Many users making changes
    - Could lead to chaos
  - Transactional systems avoid this
    - cost of ~9% performance reported on some commercial systems
    - Alas: Business customers don't think this is worthwhile ☹
- Isolation levels defined in ISO standard
  - READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE
  - Textbooks say serializable is needed
  - But immediately settle for much less ☹
- Pyrrho has always had a serialized transaction log ☺
  - Even better: Guarantees isolation by preventing conflicts

- Next: Isolation

Transaction Isolation

- Pessimistic: Lock what we plan to change
  - All commercial DBMS use this method
  - Locking things can get really complicated
  - The SERIALIZABLE isolation level results in transactions being aborted
    - On a step where serialization can no longer be guaranteed
- Optimistic: Validate when we commit
  - Then the DBMS must keep transactions isolated
  - Different row versions should be private to transaction
  - Commit will fail only if a conflicting committer gets there first
- Applications can use row versions to ensure serializability
  - Solves conflict only with other users of the same application
- Textbooks say optimistic approach should be avoided
  - We showed it can be better with the right implementation

- Next: StrongDBMS 2017

Fixing isolation: StrongDBMS

- 2017-9: A DBMS also with serialized log
  - But very strong transaction isolation
- Outperformed other DBMS in productivity
  - Transaction programming Council (TPC) benchmark test C (Clerks taking phone orders)
    - The design has many conflicts for more than 10 clerks
    - Throughput continued to increase for > 100 clerks
- But was a very simple DBMS
  - Lacking many features that people expect
- We decided to use isolation idea in Pyrrho

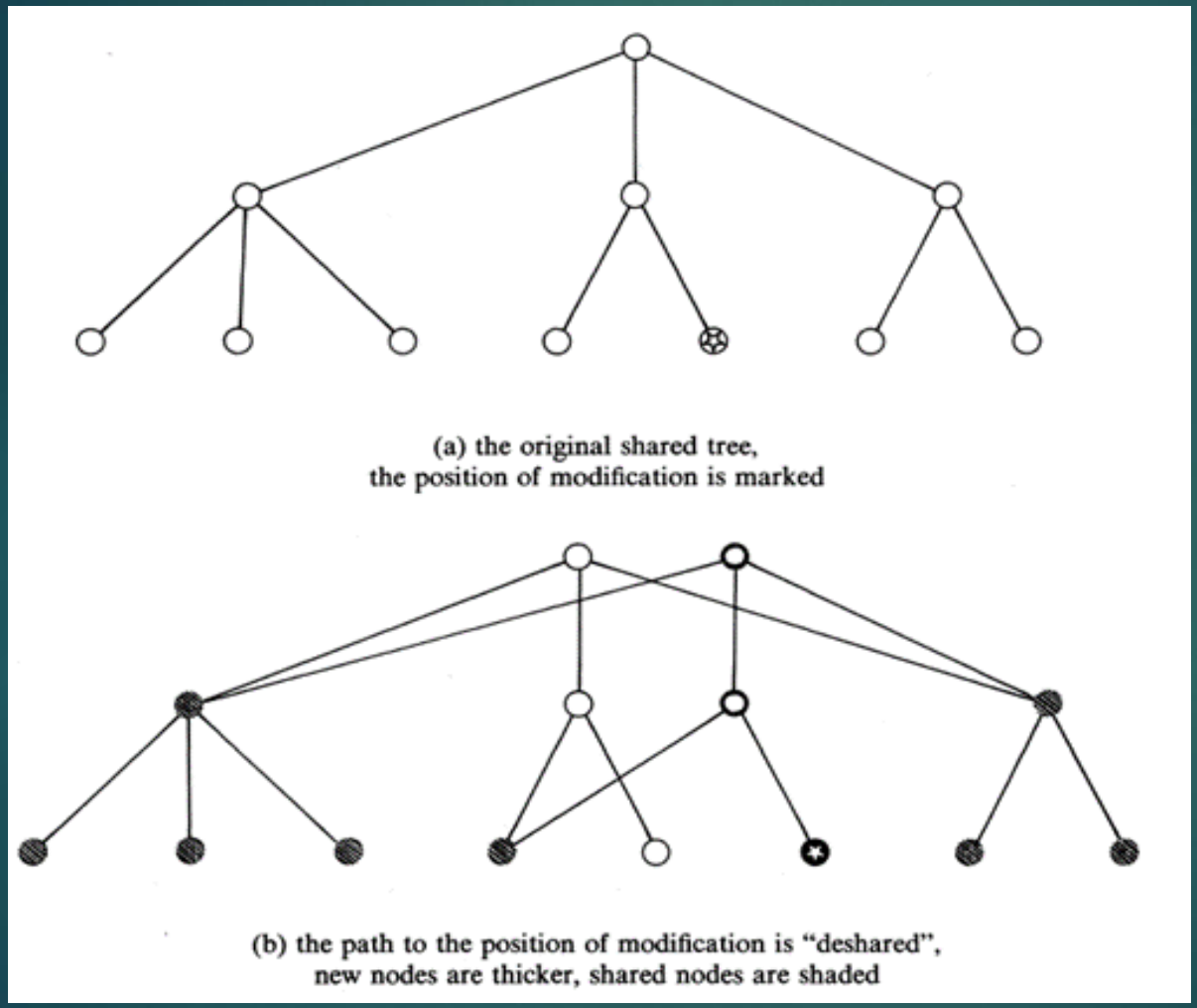
- Next: Shareable data structures

Shareable Data Structures

- StrongDB's magic ingredient was
  - SHAREABLE data structures throughout
- 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: Adding a node

When we add a node



(a) the original shared tree,
the position of modification is marked

(b) the path to the position of modification is "deshared",
new nodes are thicker, shared nodes are shaded

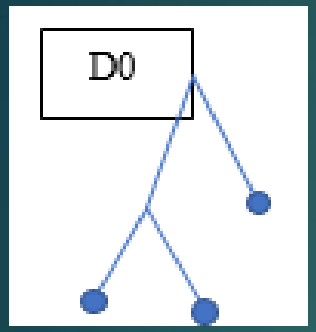
T. Krijnen, and G. L. T. Meertens, "Making B-Trees work for B". Amsterdam : Stichting Mathematisch Centrum, 1982, Technical Report IW 219/83

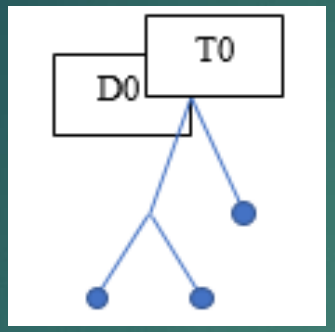
Tree structures

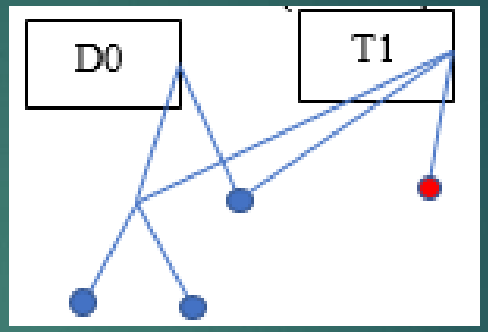
- BTree<K,V> is immutable, shareable
  - When K and V are shareable
  - Two-way traversal
  - Uses immutable bookmarks
- Database, Transaction all shareable
  - Tables, Procedures, Values shareable too
- Transaction is a private copy
  - Changes are prepared for commit step
- Database is built from the tx log

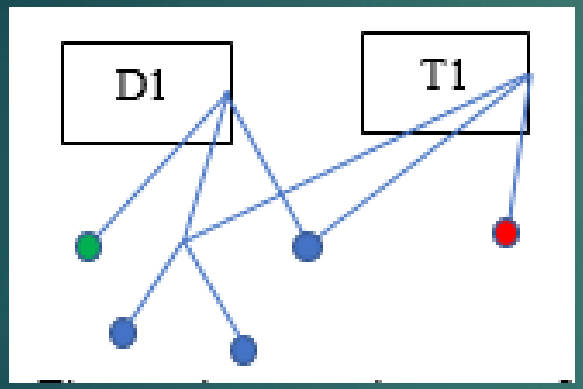
- Next: BTree and transactions

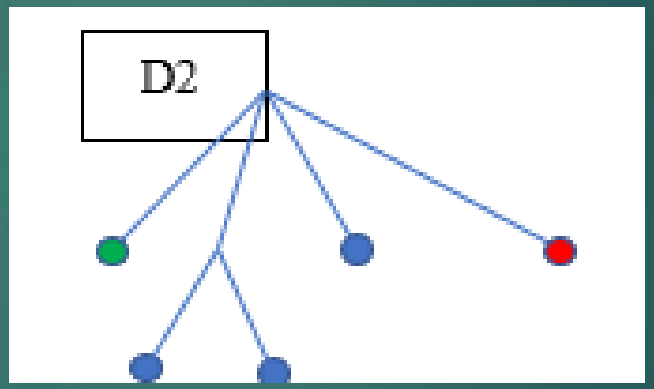
Transaction and B-Tree











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

- Next: Pyrrho outperforms

Now Pyrrho outperforms others

- In 2021, a V7 demo with good productivity
  - Productivity increasing up to 50 clerks
- A partial version of Pyrrho
  - Optimistic and Serialized, but few other features
  - And did not have quite the right structures
- By 2022, we have less productivity but
  - Fully Shareable Data Structures approach
  - Triggers, cascades, structured data type
  - Big Live Data support, virtual data warehouse

- Next: Big Live data

Big *Live* Data

- If your data originates in lots of databases
- You could copy the data centrally
  - Extract-Transform-Load/Big Data
- But, if it keeps changing this is not good
  - Much better to read just what we need now
  - And leave data where it is being maintained
- So suppose our data is remote
  - A table's rows come from different databases
    - E.g. Sales or product data from different companies
- The available data is provided as a View
  - And accessible using HTTP and JSON

- Next: A derived table

A derived table

Derived = not actually stored centrally

Columns from D's renamed and values probably transformed

CID	A	B	C	…			
D1							
D1							
D2							
D3							
D3							
D3							

(Contributors take responsibility for renaming columns and transforming data to suit us as their schemas will all be different)

- Next: Contributing DBMS

Contributing databases

- Contributors provide data in a given form
  - Hypertext Transfer Protocol (HTTP) request
  - Representational Structure Transfer (REST)
  - Java Structured Object Notation (JSON)
- They probably don't have it in this form
  - So, they create a VIEW with the right columns
  - Values probably requires some transformation
- Make it available with a given address
  - Uniform Resource Locator (URL)
  - With access permissions for our view
  - Possibly, they might allow some updates

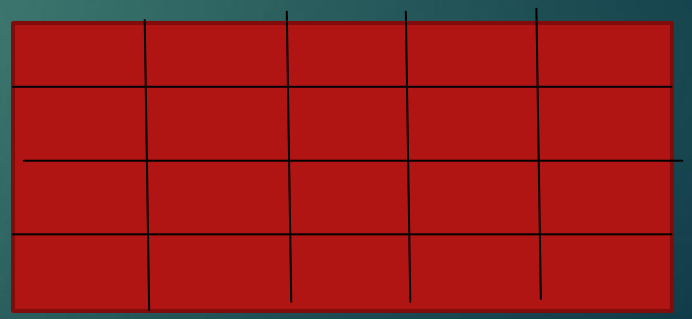
- Next: Defining contribution

Defining a contribution

- Probably, each contributor creates a VIEW
  - Out of data from one or more actual tables

CREATE VIEW (A,B,C..) AS ….

A	B	C	…			



Can identify each contributor in the result view with a contributor id CID and maybe other information

- Next: The central view

Centrally we then have

- A row type CID,..,A,B,C,..
  - The local row contains remote data
- A local table T of contributor details, URLs

T:

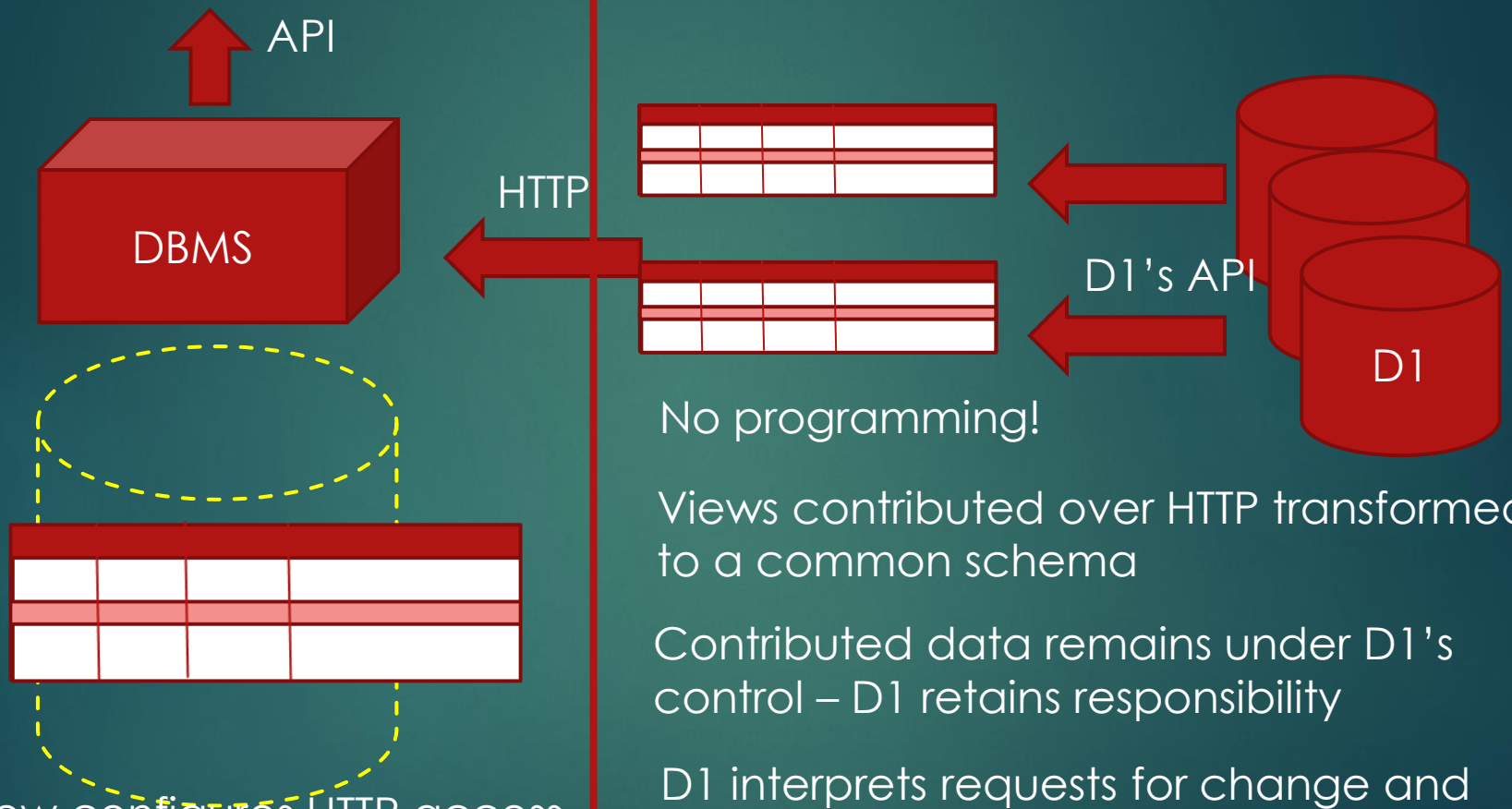
CID	…	URL
D1	…	URL for D1's data
D2	…	URL for D2's data
D3	…	URL for D3's data

CREATE VIEW V OF (CID..,A,B,C..) AS GET USING T

- OF clause gives V's row type (specifying column data types)
  - Includes all columns from T except the last (the URL)
  - The remaining columns specify the data from the remote view

- Next: Dividing responsibility

Division of responsibility



No programming!

Views contributed over HTTP transformed to a common schema

Contributed data remains under D1's control – D1 retains responsibility

D1 interprets requests for change and inverts the transformations if it can

View configures HTTP access

Change request sent to D1,..

- Next: View-mediated access

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]
```

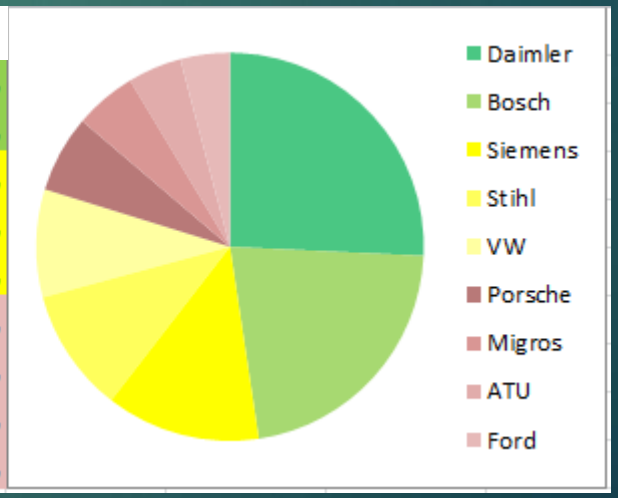
- Designed for filtering by item
  - To discourage retrieval of the entire table

- Next: 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%

	ABC	Customer	Sales	accumul.S.	... %
	A	Daimler	20000	20000	26%
t1 ->	A	Bosch	17000	37000	48%
	B	Siemens	10000	47000	60%
	B	Stihl	8000	55000	71%
t2 ->	B	VW	7000	62000	80%
	C	Porsche	5000	67000	86%
	C	Migros	4000	71000	91%
	C	ATU	3500	74500	96%
	C	Ford	3200	77700	100%



- Next: SQL for ABC

Code for ABC-Analysis

- Standard Query Language (SQL)
- With a 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

```
|---|--------|--------|-------|
|ABC|CUSTOMER|SALES   |SHARE  |
|---|--------|--------|-------|
|A  |Daimler |20000,00|25.74 %|
|A  |Bosch   |17000,00|21.88 %|
|B  |Siemens |10000,00|12.87 %|
|B  |Stihl   |8000,00 |10.30 %|
|B  |VW      |7000,00 |9.01 % |
|C  |Porsche |5000,00 |6.44 % |
|C  |Migros  |4000,00 |5.15 % |
|C  |ATU     |3500,00 |4.50 % |
|C  |Ford    |3200,00 |4.12 % |
|---|--------|--------|-------|
SQL>
```

- Next: No query rewriting

No query rewriting

- Consider the <select list> concept in View
- If it contains 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?
- ARegister 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

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

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

```
http://localhost:8180/DB/DB select (SUM(E)+CHAR_LENGTH(F)),F from t group by
HTTP POST /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
HTTP POST /DC/DC
select (SUM(E)+CHAR_LENGTH(F)),F from u group by F
Returning ETag: "23,-1,159"
--> 3 rows
```

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

```
SQL> select sum(e)+char_length(f),f  from ww group by f
|----|-----|
|Col0|F    |
|----|-----|
|11  |Ate  |
|9   |Five |
|8   |Four |
|11  |Sechs|
|9   |Six  |
|8   |Three|
|8   |Vier |
|----|-----|
SQL>
```

- 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: Example

This is a simple case

- In this case, there is just one hidden field
  - The register for SUM(E)
  - And the values of F don't overlap ☹
- DB returned 4 rows, which contained


```
{{"Col0": 11, "F": "Sechs", "$#9": {"0": 6}}}
{{"Col0": 9, "F": "Six", "$#9": {"0": 6}}}
{{"Col0": 8, "F": "Three", "$#9": {"0": 3}}}
{{"Col0": 8, "F": "Vier", "$#9": {"0": 4}}}
```


- DC returned 3 rows


```
{{"Col0": 11, "F": Ate", "$#9": {"0": 8}}}
{{"Col0": 9, "F": "Five", "$#9": {"0": 5}}}
{{"Col0": 8, "F": "Four", "$#9": {"0": 4}}}
```


- Next: Transactions and REST

Transactions and REST

- All data needs a single transaction master
  - Because of the two-army problem
- Transactions start from one database
  - Called the local database (i.e. local server)
  - There is no way to address a remote object directly
- Some fields may come from remote views
  - Possibly updatable via REST over HTTP1.1 (safe)
  - At most one remote update can be allowed
- When the local commit is called
  - Local database locked, validation performed
  - The single remote update is done via HTTP1.1
  - And then the local commit can complete/unlock

- Next: Object-Orientation

REST and Object-Orientation

- Pyrrho has a Versioned API
  - Some local base table rows are entities
  - Pyrrho generates a Versioned class
    - With navigation properties like Java Persistence, LINQ
- Support for versions is built in to the DBMS
  - Connections may have different row versions
    - A versioned object is always for a particular connection
  - Some entity fields may be remote (updatable via REST)
- Versioned class `C` has `Get(), Put(), Post(), Delete()`
- Fetch Versioned objects from a connection:
  - `C[] Get<C>(w)` and variants
    - `C[] FindAll<C>(), C FindOne<C>(key), C[] FindWith<C>(where)`
- Guaranteed transaction-safe
  - Allows long serialized transactions

- Next: Other DBMS

Future Work: 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
  - Simplify distributed problem with RESTView

- Next: Conclusions

Conclusions

- This research provides new DBMS tools
  - Shareable data structures library
- Serialized transactions
- Optimistic: no client locking of data
  - Better suited to needs of web applications
- Big Live Data implementation
  - Providing better real-time owned behavior
  - Optimized for aggregations of remote views
- Versioned API for transaction-safe apps

- Next: Links

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., Laux, F.: Reconsidering Optimistic Algorithms for Relational DBMS, DBKDA 2020

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

Krijnen, T., Meertens, G. L. T.: "Making B-Trees work for B". Amsterdam : Stichting Mathematisch Centrum, 1982, Technical Report IW 219/83