ADAPTIVITY IN DATABASE KERNELS

Adaptive Indexing: Self tuning access methods

Javam Machado Elvis Teixeira Paulo Amora

25 de agosto de 2017

Universidade Federal do Ceará - UFC

Universidade Federal do Ceará

SUMARY

- RECAP
  - New Problems
  - Adaptive vs Offline
- Database Cracking
  - Adaptive index for column stores
- Adaptive merging
  - Adaptive index for tuple based storage
- Concurrency

RECAP

Up to date data

No workload knowledge

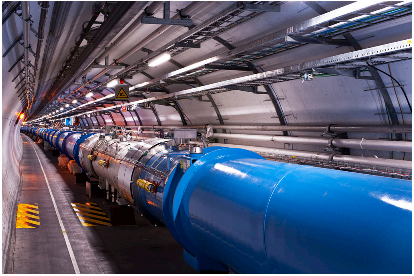


Figure: The Large Hadron Collider

Fast and large data analysis strategies:

Horizontal scalability

Specialized data models

Eventual consistency

Figure: NoSQL DBMS

Scalable

Comodity hardware

Map Reduce

Unstructured data

Figure: Apache Hadoop

Heterogeneous

Social

Autonomous

Figure: SETI @ Home

What about DBMS?

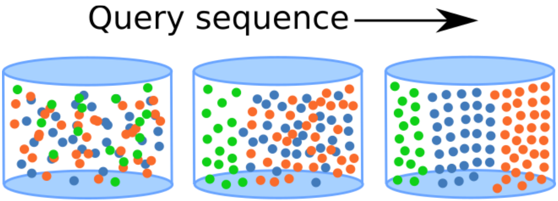
Offline indexes

- Require decisions on what to index
- One step operation (CREATE INDEX, DROP INDEX)
- Changes in workload demand rebuild

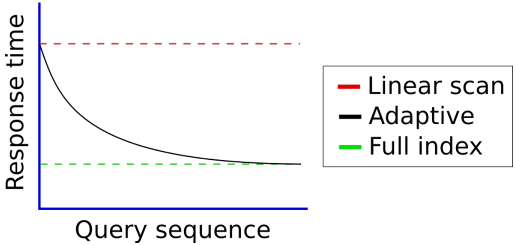
Adaptive indexes

- Physical design is tuned by incremental actions
- Changes occur in response to current query
- Changes in workload are naturally handled

ADAPTIVE INDEXING CHANGES PHYSICAL LAYOUT



Response times are expected to decrease from the level of full scans ($O(N)$) to near the level of a binary search ($O(log(N))$)



DATABASE CRACKING

- Developed for column stores (MonetDB)
- Partitions an attribute at each query
- In memory column copy and supporting AVL tree
- Low initialization cost

select ... where A >= 6;

2
0
1
3
4
9
6
8
7
5

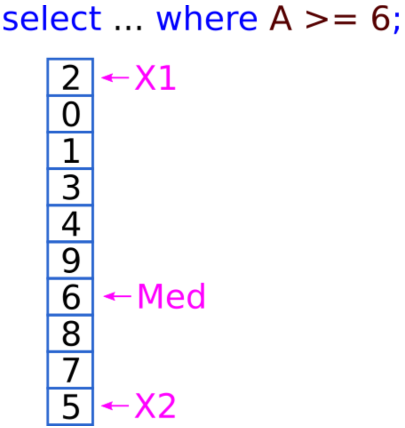
```
algorithm CrackInTwo(Low, High, Med)
    x1 := point at position Low
    x2 := point at position High
    while position(x1) < position(x2) do
        if value(x1) < Med then
            x1 := point at next position
        else
            while value(x2) >= Med and
            position(x2) > position(x1) do
                x2 := point at previous position
            end while
            Exchange(x1, x2)
            x1 := point at next position
            x2 := point at previous position
        end if
    end while
```

Idreos et. al. 2007 - Database Craking

```
algorithm CrackInTwo(Low, High, Med)
    x1 := point at position Low
    x2 := point at position High
    while position(x1) < position(x2) do
        if value(x1) < Med then
            x1 := point at next position
        else
            while value(x2) >= Med and
            position(x2) > position(x1) do
                x2 := point at previous position
            end while
            Exchange(x1, x2)
            x1 := point at next position
            x2 := point at previous position
        end if
    end while
```

Idreos et. al. 2007 - Database Craking

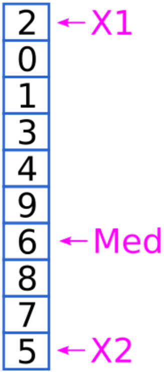
CRACKING COLUMN



```
algorithm CrackInTwo(Low, High, Med)
    x1 := point at position Low
    x2 := point at position High
    while position(x1) < position(x2) do
        if value(x1) < Med then
            x1 := point at next position
        else
            while value(x2) >= Med and
            position(x2) > position(x1) do
                x2 := point at previous position
            end while
            Exchange(x1, x2)
            x1 := point at next position
            x2 := point at previous position
        end if
    end while
```

Idreos et. al. 2007 - Database Craking

select ... where A >= 6;



CRACKING COLUMN

select ... where A >= 6;

UNIVERSIDADE FEDERAL DO CEARÁ

```
algorithm CrackInTwo(Low, High, Med)
    x1 := point at position Low
    x2 := point at position High
    while position(x1) < position(x2) do
        if value(x1) < Med then
            x1 := point at next position
        else
            while value(x2) >= Med and
            position(x2) > position(x1) do
                x2 := point at previous position
            end while
            Exchange(x1, x2)
            x1 := point at next position
            x2 := point at previous position
        end if
    end while
```

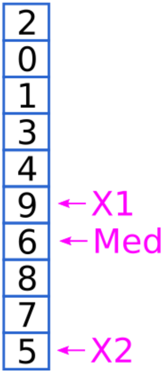
Idreos et. al. 2007 - Database Craking

select ... where A >= 6;

```
algorithm CrackInTwo(Low, High, Med)
    x1 := point at position Low
    x2 := point at position High
    while position(x1) < position(x2) do
        if value(x1) < Med then
            x1 := point at next position
        else
            while value(x2) >= Med and
            position(x2) > position(x1) do
                x2 := point at previous position
            end while
            Exchange(x1, x2)
            x1 := point at next position
            x2 := point at previous position
        end if
    end while
```

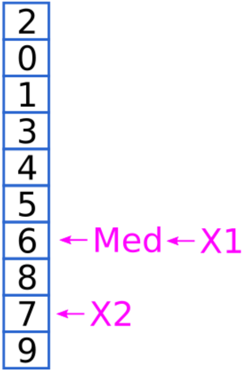
Idreos et. al. 2007 - Database Craking

select ... where A >= 6;




select ... where A >= 6;
2
0
1
3
4
5
6
8
7
9
←X1
←Med
←X2

select ... where A >= 6;

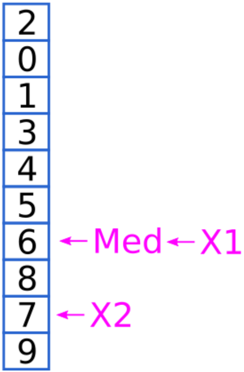


UNIVERSIDADE FEDERAL DO CEARÁ

```
algorithm CrackInTwo(Low, High, Med)
    x1 := point at position Low
    x2 := point at position High
    while position(x1) < position(x2) do
        if value(x1) < Med then
            x1 := point at next position
        else
            while value(x2) >= Med and
            position(x2) > position(x1) do
                x2 := point at previous position
            end while
            Exchange(x1, x2)
            x1 := point at next position
            x2 := point at previous position
        end if
    end while
```

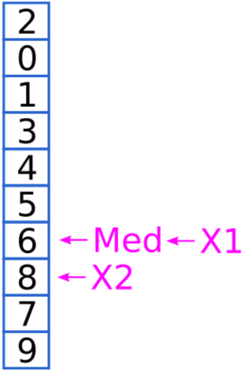
Idreos et. al. 2007 - Database Craking

select ... where A >= 6;

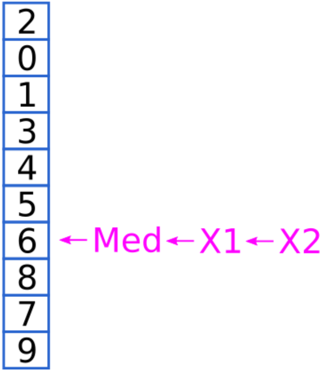


CRACKING COLUMN

select ... where A >= 6;

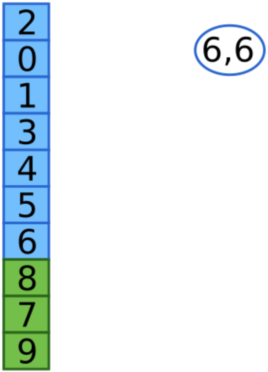


select ... where A >= 6;

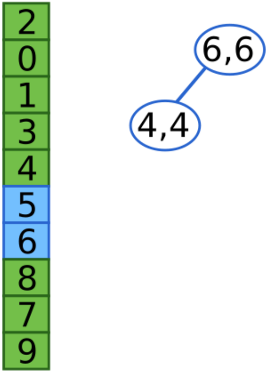


Partitions are stored in a tree structure (cracker index)

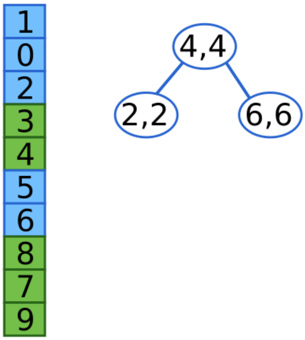
Partitions are stored in a tree structure (cracker index)



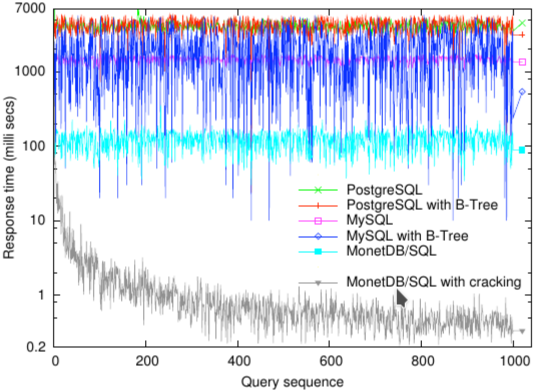
More queries - more partitions - smaller pieces scanned



More queries - more partitions - smaller pieces scanned



Database cracking - response times



Idreos et. al. 2007 - Database Cracking

A histogram for free [1]

Column partitions contain information on the distribution of the data attribute. i. e. they tell how many records lie in the given range.

[1] Idreos et. al. 2007 - Database Craking

Stochastic cracking

- Partition ranges are not equal to query ranges
- Adds a random component to cracking
- Eventually cracks big partitions

Holistic indexing

- Idle CPU cores are used to perform cracks
- Select operators still perform cracks
- Holistic cracks are performed on the biggest partitions

ADAPTIVE MERGING

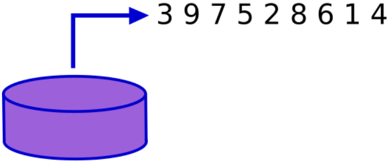
Relational systems are typically stored in disk

B-tree based structures are suitable for block storage

Full sorting may be prohibitive (time)

And demands prior index selection (workload knowledge)

PARTITIONED B TREES



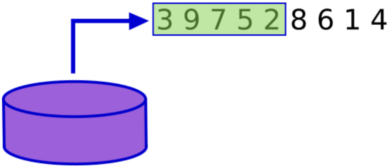


Figure: Collect run

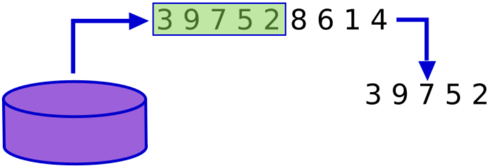


Figure: Collect run

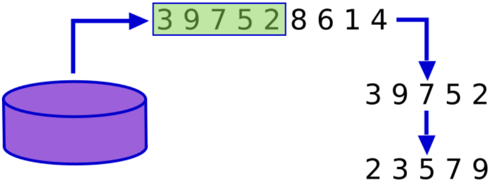


Figure: Sort run

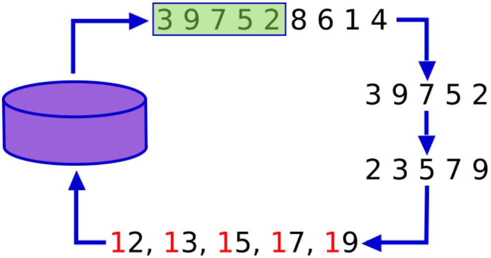


Figure: Add partition key

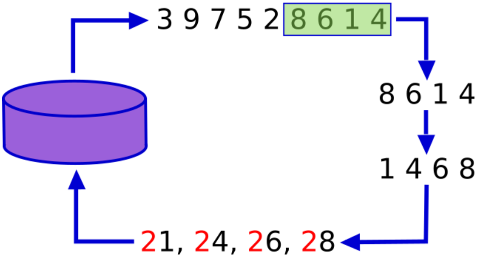


Figure: Repeat for other partitions

12, 13, 15, 17, 19, 21, 24, 26, 28

Figure: Final sorted data

Structure creation

- Runs become the data in the leaf level of a B+ tree
- A bulk load procedure is used to build the tree

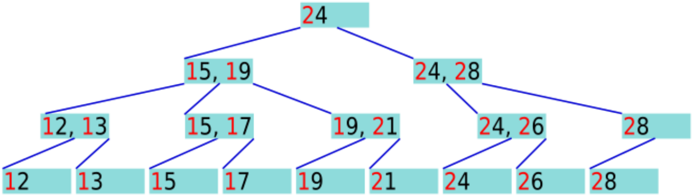


Figure: Complete tree

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

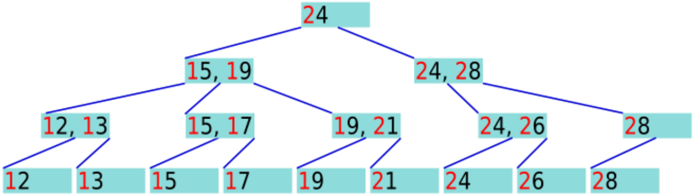


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

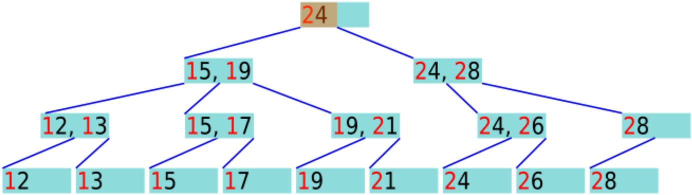


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

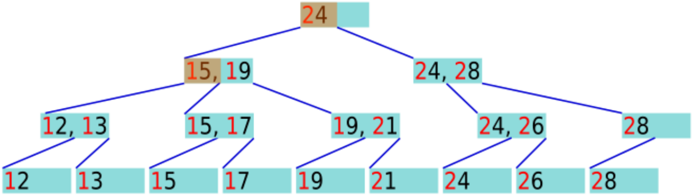


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

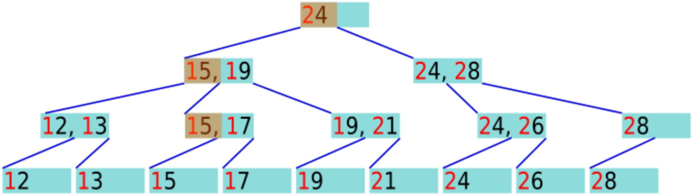


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

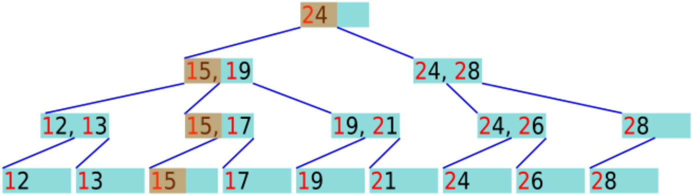


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

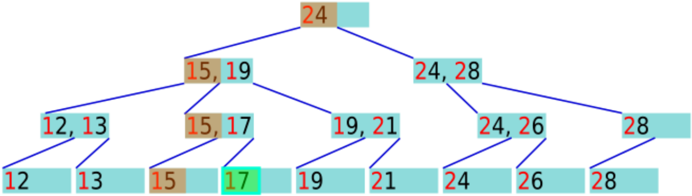


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

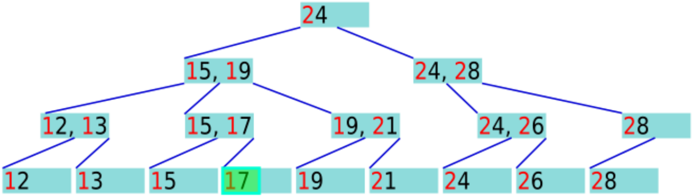


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

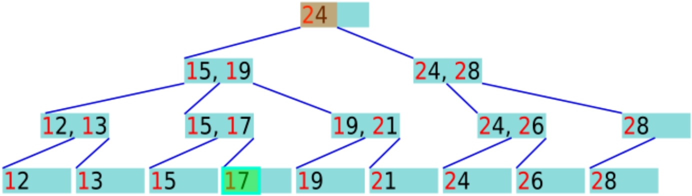


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

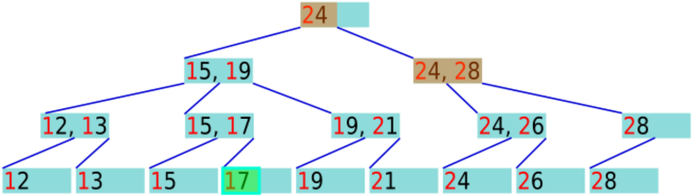


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

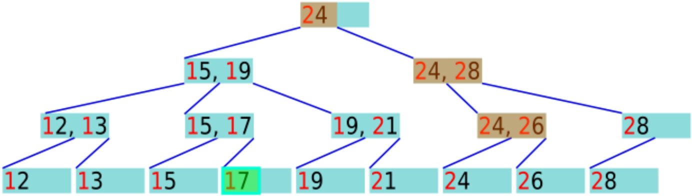


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

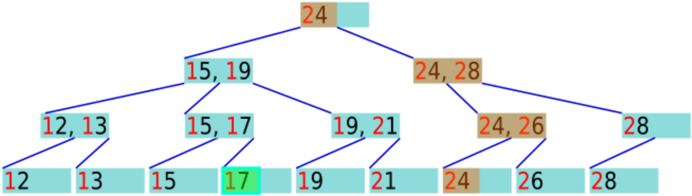


Figure: Answering a query

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

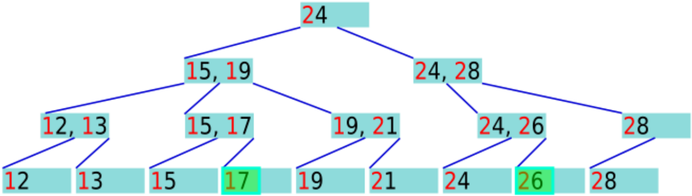


Figure: Answering a query

Each query walks the tree and move the qualifying tuples to the final partition

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

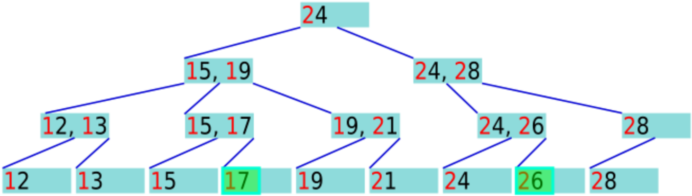


Figure: Adaptive Merging

SELECT * FROM t WHERE t.A > 5 AND t.A <= 7;

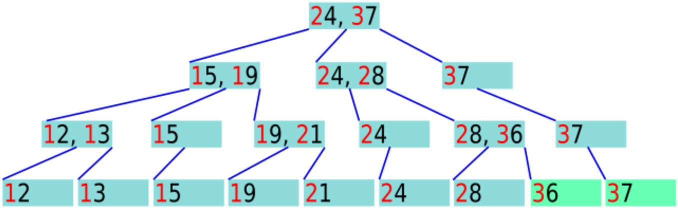


Figure: Short Query Ranges

Adaptive Merging - overhead per query

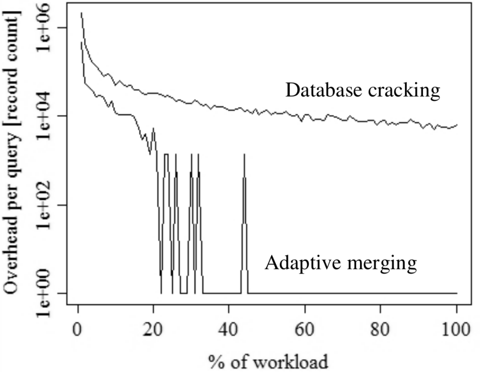


Figure: Short Query Ranges

Grafe et. al. 2010 - Self-selecting, self-tuning incrementally optimized indexes

UNIVERSIDADE FEDERAL DO CEARÁ

Adaptive Merging - overhead per query

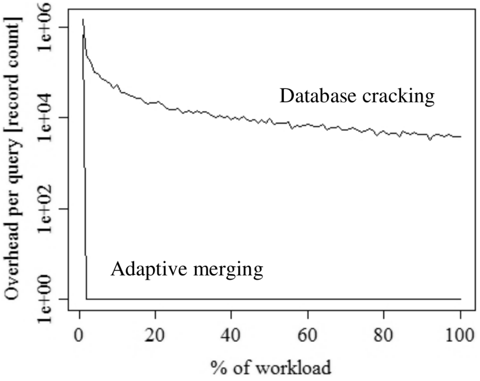


Figure: Long Query Ranges

Grafe et. al. 2010 - Self-selecting, self-tuning incrementally optimized indexes

CONCURRENCY

The problem

Updating index structures while processing queries requires concurrency control and the system may incur additional lock contention

Index structure VS index contents [2]

- Index logical contents do not change
- Index refinement is not transactional
- Lightweight latches instead of locks

[2] Graefe et. al. 2012 - Concurrency Control for Adaptive Indexing

Locks VS Latches

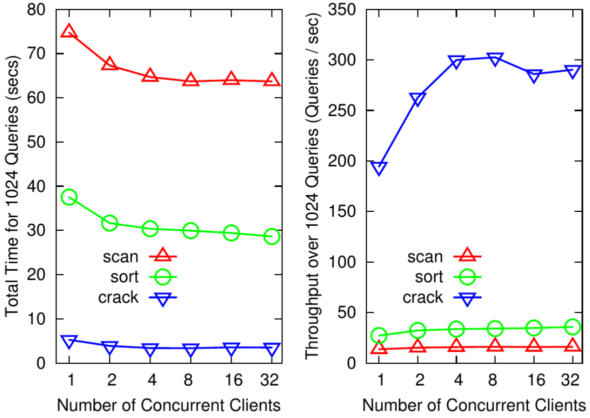
	Locks	Latches
Separate	Transactions	Threads
Protect	DB Content	In-memory data
During	Entire Transactions	Critical sections

Incremental granularity of locking [3]

- Increasingly smaller key ranges affected
- Conflicts can be avoided

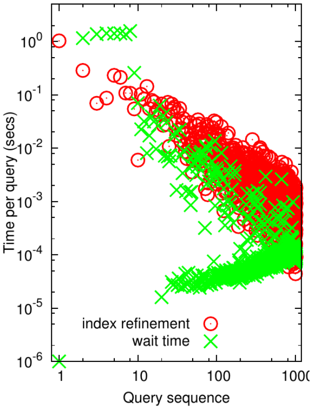
[3] Graefe et. al. 2012 - Concurrency Control for Adaptive Indexing

Throughput



Graefe et. al. 2012 - Concurrency Control for Adaptive Indexing

Wait time



Graefe et. al. 2012 - Concurrency Control for Adaptive Indexing

AI/ML guided layout optimization

- Incremental physical layout tuning enables learning
- Current request X Workload pattern
- Workload forecasting (tune in anticipation)

Flexible physical design

Uses workload pattern recognition

Fits modern query processing needs

F. Funke et. al. - 2012. Compacting Transactional Data in Hybrid OLTP&OLAP Databases

H. Lang et. al. - 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation

I. Alagiannis et. al. - 2014. H2O: A Hands-free Adaptive Store

J. Arulraj et. al. - 2016. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads

Graefe - 2010. Self-selecting, self-tuning, incrementally optimized indexes

Idreos - 2007. Database Cracking

Questions?