Thomas Kissinger, Peter Volk, Benjamin Schlegel

Forschungskolleg Datenbanken
Modern-Hardware-Support for DBMS
DEXTER / EAST@HAEC
Parallel In-Memory Indexing and Direct Query Processing on Prefix Trees
... and SFB 912: HAEC
Hardware-Level Trends

- **Shifting Memory Hierarchy**
  - In-memory databases/indexing
  - Cache-awareness
  - Other Block/MTU sizes
  - Sequential access still faster than random access

- **More and more Cores**
  - Clock rate got stuck
  - Massive parallelism
  - Optimized software required

- **Modern Hardware Architectures**
  - SIMD instructions
  - Multiple memory channels
  - SMP changes to NUMA
  - Heterogeneous Hardware (GPUs, Co-processors ...)
  - Adaptive Hardware (FPGAs, ...)

- **DEXTER: Parallel In-Memory Indexing and Query Processing on Prefix Trees**
Application-Level Trends

Evolution of DWH Applications

- **Advanced Analytics**
  - Sophisticated statistical models
  - Machine learning
  - Mixed workloads

- **Operational BI**
  - High update rates
  - Transactional workloads

**Reporting**
- What did happen?
- Create reports with pre-defined queries.

**Analysis**
- Why did it happen?
- Increasing number of ad-hoc queries.

**Advanced Analytics**
- What will happen?
- Extension of the analytical model (e.g. Forecasting).

**Operational BI**
- What happens right now?
- Continuous streams of ad-hoc queries and propagated updates.
State-of-the-Art System Architecture

- **SAP HANA, C-Store, ...**

![Diagram showing write-optimized temporary store, periodic merge, and read-optimized column stores for ad-hoc queries.](image)

**Shortcomings**
- Poor point query support (e.g. Advanced Analytics)
- Additional query of temporary store necessary
- Periodic merge rebuilds column store and stalls queries in the meantime

- **In-memory**
  - About 30GB/s theoretical peak read performance
- **Highly parallel**

Dipl.-Ing. Thomas Kissinger

DEXTER: Parallel In-Memory Indexing and Query Processing on Prefix Trees
Center for Information Services and High Performance Computing (ZIH)
Measurement at June 20, 2008

Goal: Minimizing Energy by Multi-Layer SW/HW Adaptivity
**HAEC Box**

**Optical Interconnect**
- adaptive analog/digital circuits for e/o transceiver
- embedded polymer waveguide
- packaging technologies (e.g. 3D stacking of Si/III-V hybrids)
- 90° coupling of laser

**Radio Interconnect**
- on-chip/on-package antenna arrays
- analog/digital beamsteering and interference minimization
- 100Gb/s
- 100-300GHz channel
- 3D routing & flow management
Highly Adaptive Energy-Efficient Computing

Direct Stream Processing
- HAEC Box
  - Sensor Data Stream
  - Input Buffer
  - Load Shedding
  - Load Balancing
- Heterogeneous Stream Processing Lanes

Direct Query Processing
- Massive parallel operators
- Fast working copies with help of virtual memory management
- n-ary operators

Upper Applications
- Core Indexing Structure
  - Generalized prefix tree
  - In-Memory
  - Balanced read/write performance

Flash-optimized Storage System
DEXTER Project

(Dresden Index for Transactional Access on Emerging Technologies)

- Transactional Index, optimized for mixed queries and high update rates
- Sponsored by SFB 912: HAEC (Highly Adaptive Energy-Efficient Computing)

Matthias Böhm
Thomas Kissinger
Peter B. Volk
Ulrike Fischer
Benjamin Schlegel
Dirk Habich

© Dipl.-Ing. Thomas Kissinger
DEXTER Project Map

**ACM SIGMOD**

**Programming Contest**

2nd place 2009
indexing system for main memory data

1st place 2011
high-throughput main-memory index
which is made durable using a flash-based SSD

**High point query throughput**

**High update rates**

**Direct Query Processing**

**Core Indexing**

DEXTER: Parallel In-Memory Indexing and Query Processing on Prefix Trees


FAST: fast architecture sensitive tree search on modern CPUs and GPUs

- **SIGMOD 2010**
  
  *Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt and Pradeep Dubey*
  
  • Memory hierarchy optimized binary tree
  • Read-optimized index structure
  • 51M reads/s, but only 10 updates/s for a tree of 64M keys

PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

- **VLDB 2011**
  
  *Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish and Pradeep Dubey*
  
  • B+-Tree based index
  • Synchronous batch updates to avoid latches

Both index structures suffer from a poor update performance
Core Indexing Structure: Prefix Tree

- Only one 64bit access per node
- Deterministic path
- No Balancing

Static Prefix Length: \( k' = 4 \)
Prefix Tree: Configurations

Static Prefix Length: $k' = 1$

000000000011010111
000000000011010111

flatter tree $\rightarrow$ Less memory transfers

deep tree $\rightarrow$ More memory transfers / Better memory utilization

Variable prefix length on tree level or node level possible
Parallel In-Memory Indexing

Two important dimensions for parallel in-memory indexing

- **Partitioning**: No vs. Yes
- **Batching**: No vs. Yes

<table>
<thead>
<tr>
<th>Partitioning</th>
<th>Batching</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>FAST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>PALM</td>
<td>DEXTER Batched Reads</td>
<td>DEXTER BUZZARD</td>
</tr>
</tbody>
</table>

- **Direct Processing**
- **Synchronization**
- **Independent Data**
- **Indirect Processing**

- **Low latency**
- **High Throughput**
- **Increased Latency**
Two important dimensions for parallel in-memory indexing

<table>
<thead>
<tr>
<th>No Partitioning</th>
<th>Batching</th>
<th>Yes Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAST</td>
<td>DEXTER</td>
<td>PALM</td>
</tr>
<tr>
<td></td>
<td>DEXTER</td>
<td>DEXTER BUZZARD</td>
</tr>
</tbody>
</table>

- + Direct Processing
- - Synchronization
- + Independent Data
- - Indirect Processing
- + Low latency
- - Increased Latency
- + High Throughput
Parallel DEXTER

- Heavyweight Latches
  - Mutex/Futex
  - Spinlock

Read-Copy-Updates (RCU) - aware Memory Manager
Parallel DEXTER

- Heavyweight Latches
  - Mutex/Futex
  - Spinlock

- Lightweight Atomics
  - Compare-and-Swap (CAS)

### Lightweight Atomics

- Compare-and-Swap (CAS)

### Heavyweight Latches

- Mutex/Futex
- Spinlock

Read-Copy-Updates (RCU) - aware Memory Manager
Parallel DEXTER: Evaluation

- Intel i7-2600 (4 cores @3.4GHz, 2 memory channels, 16GB DDR3 @1333MHz)

Graph showing the relationship between million Keys and Million Operations/s.
Parallel DEXTER: Evaluation

- Intel i7-2600 (4 cores @3.4GHz, 2 memory channels, 16GB DDR3 @1333MHz)

![Graph showing 100% Updates]
Parallel DEXTER: Evaluation

- Intel i7-2600 (4 cores @3.4GHz, 2 memory channels, 16GB DDR3 @1333MHz)

![Graph showing performance of DEXTER with 100% updates and reads.](image)
Parallel DEXTER: Evaluation

- Intel i7-2600 (4 cores @3.4GHz, 2 memory channels, 16GB DDR3 @1333MHz)
- 20M Keys (32bit)
Parallel DEXTER: Evaluation

- Intel i7-2600 (4 cores @3.4GHz, 2 memory channels, 16GB DDR3 @1333MHz)
- 20M Keys (32bit)

![Graph showing 100% reads vs. number of threads]
> Parallel DEXTER: Evaluation

- Intel i7-2600 (4 cores @3.4GHz, 2 memory channels, 16GB DDR3 @1333MHz)
- 20M Keys (32bit)

![Graph showing the performance of DEXTER with increasing number of threads for 100% Updates and 100% Reads.](image-url)
Two important dimensions for parallel in-memory indexing

<table>
<thead>
<tr>
<th>Partitioning</th>
<th>Batching</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Direct Processing
- Synchronization
- Independent Data
- Indirect Processing

- Low latency
- Increased Latency
- High Throughput
Parallel DEXTER: Batched Reads

- Reads some nodes (especially top-level nodes) only once
- Prefetching possible
**Parallel DEXTER: Batched Reads: Eval**

- **Intel i7-2600 (4 cores @3.4GHz, 2 memory channels, 16GB DDR3 @1333MHz)**
- **20M keys (32bit), 256-batch size**

---

![Graph showing the relationship between the number of threads and million operations per second (Million Operations/s)].

The graph illustrates the performance of DEXTER with varying numbers of threads. As the number of threads increases, the million operations per second (Million Operations/s) also increase.

- **Y-axis**: Million Operations/s
- **X-axis**: Number of Threads

---

© Dipl.-Ing. Thomas Kissinger | DEXTER: Parallel In-Memory Indexing and Query Processing on Prefix Trees | 26
Parallel DEXTER: Batched Reads: Eval

- Intel i7-2600 (4 cores @3.4GHz, 2 memory channels, 16GB DDR3 @1333MHz)
- 20M keys (32bit), 256-batch size

![Graph showing batched reads vs. number of threads]

- Reads
- Batched Reads

Million Operations/s vs. # of Threads
- Intel i7-2600 (4 cores @3.4GHz, 2 memory channels, 16GB DDR3 @1333MHz)
- 20M keys (32bit), dynamic batch size
### Parallel In-Memory Indexing

**Two important dimensions for parallel in-memory indexing**

<table>
<thead>
<tr>
<th>Partitioning</th>
<th>Batching</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>FAST</td>
<td></td>
<td>DEXTER Buzard</td>
</tr>
<tr>
<td>Yes</td>
<td>PALM</td>
<td></td>
<td>DEXTER Buzard</td>
</tr>
</tbody>
</table>

- **Low latency**
- **High Throughput**
- **Direct Processing**
- **Indirect Processing**
- **Independent Data**
- **Increased Latency**
- **Synchronization**
Incoming Requests

- Thread Mediation Layer
- Decoupling

- Partitioned Prefix Tree
  - Adaptive partitioning
  - No synchronization needed

- Memory Mapping
  - Map tree partition to local memory on NUMA systems
> Outline

- **INTRODUCTION AND TRENDS**

- **DEXTER PROJECT**

- **CORE INDEXING**

- **DIRECT QUERY PROCESSING**
  - Overview
  - Memory Mirroring
  - Range Select

- **CONCLUSIONS**
DEXTER Project Map

- Exploit prefix trees for efficient analytical queries
- Database operations on prefix trees

Direct Query Processing

Core Indexing

High point query throughput

High update rates
Memory Mirroring

- **Idea:** Utilize functionality of the underlying hardware and OS to get fast copies of prefix trees for destructive DB operators

- **Original Prefix Tree**
  - `memcpy`
  - Copies physical memory
  - Fully independent data

- **mmapped shared memory**
  - References old VMM
  - Integrates into address space

- **fork**
  - Copies VMM
  - Complex to control

- **memcpy**
  - Copies physical memory
  - Fully independent data
Memory Mirroring

- **Idea:** Utilize functionality of the underlying hardware and OS to get fast copies of prefix trees for destructive DB operators

  - **Original Prefix Tree**

    - **memcpy**
      - Copies physical memory
      - Fully independent data

    - **fork**
      - Copies VMM
      - Complex to control

  - **mmapped shared memory**
    - References old VMM
    - Integrates into address space

    << ms

    seconds

    Memory

    VMM

    VMM

    Memory

    VMM

    VMM
Memory Mirroring

- **Idea**: Utilize functionality of the underlying hardware and OS to get fast copies of prefix trees for destructive DB operators

- **Original Prefix Tree**

- **memcpy**
  - Copies physical memory
  - Fully independent data

- **fork**
  - Copies VMM
  - Complex to control

- **mmapped shared memory**
  - References old VMM
  - Integrates into address space

- **seconds**

- **ms**

- **<< ms**
Range Selection

- Example for a destructive unary DB operator
- Range selection: $\sigma_{111 \leq A \leq 57344}$

- Prune beyond left key path
- Prune beyond right key path
Themen

**Flash-optimized Storage System**

- Direct Stream Processing
  - Core Structure Optimization
    - Tail Compaction (Patricia)
    - Node Compression
    - Dynamic Prefix Lengths

- Multidimensional Indexes

- Core Indexing Structure
  - Generalized prefix tree
  - In-Memory
  - Balanced read/write performance

- Upper Applications
  - Sensor Data Stream
  - Input Buffer
  - Load Shedding
  - Load Balancing
  - Heterogeneous Stream Processing Lanes

- Flash-optimized Storage System
**Themen**

- Flash-optimized Storage System
- Direct Stream Processing
  - HAEC Box
    - Sensor Data Stream
    - Input Buffer
  - Upper Applications
    - Sensor Data Stream
    - Input Buffer
    - Load Shedding
    - Load Balancing
    - Heterogeneous Stream Processing Lanes
- Direct Query Processing
  - Massive parallel operators
  - Fast working copies with help of virtual memory management
  - n-ary operators
- Core Indexing Structure
  - Generalized prefix tree
  - In-Memory
    - Balanced read/write performance

- Find the best Storage Format
  - Row/Column – Store?
  - Prefix Tree as Primary Index?
  - Hybrid Solutions?
Themen

Direct Stream Processing
- HAEC Box
  - Sensor Data Stream
  - Input Buffer
  - Load Shedding
  - Load Balancing

Direct Query Processing
- Massive parallel operators
- Fast working copies with help of virtual memory management
- n-ary operators

Upper Applications
- Sensor Data Stream
- Input Buffer
- Load Shedding
- Load Balancing

Core Indexing Structure
- Generalized prefix tree
- In-Memory
- Balanced read/write performance

- Operator Design & Implementation
  - Massive parallel
  - Operates on Prefix Trees

- Intermediates Recycling

- DB/OS: VMM on L4
Themen

Direct Stream Processing

- HAEC Box
  - Sensor Data Stream
  - Input Buffer
  - Load Shedding
  - Load Balancing
  - Heterogeneous Stream Processing Lanes

Direct Query Processing

- Massive parallel operators
- Fast working copies with help of virtual memory management
- n-ary operators

Upper Applications

- Load Shedding & Balancing
  - Intra-Operator & inter-Lane parallelism
  - Decide when to drop data
  - Push data to a specific processing lane

- Dynamic pre-aggregation based on Application Requests

Flash-optimized Storage System

- In-Memory
- Balanced read/write performance
GPU basierte unterstützte Anfrageverarbeitung
Warum GPUs

Ursprünglich entwickelt um „Bilder“ zu malen
Hohe Anforderungen von Film und Spiele Industrie nach mehr Rechenleistung

Vor 2007
- Strickte Verarbeitungspipeline
- Keine freie Programmierung

Seit ~2007
- „Freie“ Programmierung
- Von Nvidia: CUDA
- OpenCL Konsortium zu Standardisierung der Entwicklung

Ziel
- Effiziente Benutzung von GPUs als „Co“ Prozessor für Datenverarbeitung
Wo liegt die Herausforderung?

Viele einfache Prozessoren
Wenig (bis 6 GB) globaler Speicher
Separater Transfer von HS zu GPU notwendig
ABER!
Sehr hohe Bandbreite zum Speicher
„Programmierbarer“ cache
  - Kann effizienter genutzt werden als traditioneller cache
Zugriffs Muster zum Speicher haben sehr hohen Einfluss auf den Durchsatz

~20GB/s
~6GB/s
~150GB/s
~1TB/s
Arten der Parallelität

Anfrage graph einer SQL anfrage
- Bushy trees als parallele Anfrage form
- Parallelisierung von Einzel Operatoren
- Parallele Ausführung von Operatoren

Limitierungen
- Grad der Parallelität bestimmt durch Anzahl der Joins
- Ausnutzung eines parallelen Systems erst mit vielen joins

Ansatz
- Auflösen von Abhängigkeiten zur Erhöhung der Parallelität
- Reduzierung der Kommunikation
- Shrubby trees
Mögliche Arbeiten I

Re-Optimierung während der Laufzeit

- Optimierung des Planes basiert auf Statistiken und Schätzung von Kosten der Operatoren
- Fehler in der Optimierung können zu schlechten Plänen führen
- Lösungen
  - Verbessere deine Schätzungen; sehr schwierig, da numerisch komplex und
  - Erkenne während der Laufzeit, dass dein plan doch nicht so gut ist und optimiere neu soweit es geht
- Details:
  - Erkennung von schlechter Planausführung
  - Strategieentwicklung für die Reoptimierung
  - Implementieren und experimentieren
Mögliche Arbeiten II

Entwicklung weiterer Optimierungsansätze

- Herausforderung bei der Optimierung: finde den besten plan!
- Derzeit: top down Optimierung
  - Nehme einen nicht parallelen plan und optimiere ihn

- Weiterer Ansatz: bottom up Optimierung
  - Nehme den parallelsten Plan und optimiere ihn in Richtung nicht nicht parallelen Plan

- Detail
  - Identifikation des parallelestes plan und Auswahl für Start der Optimierung
  - Entwicklung von Regeln für die Transformation
Parallel Database Primitives
Available parallelism

Super-scalar processors (Instruction-level parallelism)
- Issue more than one instruction at a time
- Up to 4 instructions on the Nehalem architecture

SIMD capabilities (Data-level parallelism)
- Data parallelism: same instruction on multiple data
- Restrictions on memory access

Multi-core (Thread-level parallelism)
- Independent threads on each core (up to 32)
- Cores share memory controller and caches

Shared multi-processor (Thread-level parallelism)
- Independent threads on each processor (up to 8)
- Memory access time is not constant (NUMA)
Database primitives 1: Searching

Efficiently search a read-optimized index
- Many application areas (databases, IR)
- Standard algorithm: binary search
- Based on comparisons
- Complexity: $\log_2(N)$

Parallel binary search
- Divide search space into $p$ partitions
- Search within each partition using binary search
- $(p-1)$ searches are unsuccessful
- Complexity: $\log_2(N/p)$

$P = 8$
$N = 1,000,000$

Speedup = 1.17x
(20/17)

[Kaldeway, 2009]
**K-ary search**

- Divide search space in each iteration into $p$ partitions
- Reduce search space to right partition
- Complexity: $\log_{(p+1)}(N)$
- Works only with DLP

\[ P = 8 \]
\[ N = 1,000,000 \]

\[ \text{Speedup} = 3.2x \]

\[ \frac{20}{6.3} \]

**Exploit other parallelism for multiple searches**

- Employ *software pipelining* (ILP)
- Independent threads (TLP)
Compression

- Many application areas (IR, databases)
- Reduces memory consumption
- Increases available bandwidth

Parallel compression (TLP)

- Divide dataset into $p$ partitions
- Process each partition using a single thread
- Speedup: $p$ times for $p$ threads

What about ILP and DLP???
Exploiting ILP
- Remove *unpredictable* branches (predication)
- Avoid data-dependent latencies

Exploiting DLP
- Restricted to chunk-load instructions (i.e., no scatter/gather support)
- Utilize byte permutation instruction

Example: Null suppression
- Removes leading zeros of an integer value
- Stores number leading zeros and *effective bytes*

**Compression**

<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>02 03 07 09</td>
<td>0A 0B 0F 80</td>
<td>80 80 80 80</td>
<td>80 80 80 80</td>
</tr>
</tbody>
</table>

Permutation mask **B7**

Uncompressed values (32-bit) Leading zeros (10,11,01,11) → **B7**

Compressed values

**Decompression**

<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 80 00 01</td>
<td>80 80 80 02</td>
<td>80 03 04 05</td>
<td>80 80 80 06</td>
</tr>
</tbody>
</table>

Inverse permutation mask **B7**

Compressed values

Uncompressed values (32-bit)
Database primitives: Set operations

Sorted-set intersection
- Heavily used operation in query processing (IR and databases)
- Classification into search-based or merge-based approaches

```
Search-based
A
B
```

```
Merge-based
A
B
```

Merge-based intersection
- Simple algorithm
- But: difficult to parallelize

Exploiting thread-level parallelism
- Partition sets into quantiles
- Threads process quantiles independently
- Load distribution is difficult
Exploiting instruction-level parallelism
- Remove branches in the main loop
- BUT: Not always beneficial
- Utilize software pipelining

Exploiting data-level parallelism
- Utilize STTNI instructions (allow 256 comparisons per 2 cycles)
- Employ speculative comparisons
**Student Topics 1: Build scalable Algorithms**

*Scalable algorithms*
- Efficient workload partitioning
- Numa-aware memory allocation strategies

*Setup*
- 4P Server (8 Xeon Cores @ 2.13GHz)
- 128GB RAM
- 250GB PCIe SSD

*Possible tasks*
- Scalable database operators (compressed data scan, aggregation, sorting, ...)
- Scalable set operations (e.g., answer search queries)
- ...and others
Accelerated Processing Units (APU)
- Combines CPU and programmable GPU
- High-speed bus architecture
- Low energy consumption

Setup
- Node: AMD Bobcat (Dual core @ 1.6GHz), 4GB RAM, 60GB SSD
- 32 Nodes connected via GB-Ethernet

Possible tasks
- Set operations optimized for APUs
- Distributed operators (scan, aggregation,...)
- Distributed search index (k-ary search)
- (Distributed mining algorithms)
- ...and others
Customized instructions
- Tailor-made for data-base primitives
- Analyse potential and costs (e.g., chip space)

Setup
- Joint work with Faculty of Electrical and Computer Engineering
- Tensilica Customizable Processor

Possible tasks
- Parallel database scans (incl. predicate evaluation, aggregation)
- Parallel compression (null suppression, delta encoding, …)
- Parallel set operations (intersection, union, difference)
Dirk Habich

Wrap-up und Organisatorisches
Ziel

- Nutzung offener Daten zur Datenanalyse durch Nichtexperten

Lösung umfasst...

- Open Data Integration
  - Semi-automatische Schemaannotation (Entity and Relationship Extraction, Dedublication etc.)
  - Ergänzt um Crowd-Sourcing-Techniken
- Open Data Exploitation
  - Keyword-basierte Datenanalyse, Mapping auf Relationen
  - Open-World-Ansatz, Integration offener Daten in lokales Schema

Ansprechpartner

- Katrin Braunschweig (katrin.braunschweig@tu-dresden.de)
- Julian Eberius (julian.eberius@tu-dresden.de)
- Maik Thiele (maik.thiele@tu-dresden.de)
Data Analytics

Generalisierter Clusteringprozess
- Kombination multipler Clusterings
- Ergebnisvisualisierung und Nutzerinteraktion
- Martin Hahmann (martin.hahmann@tu-dresden.de)

Selbstoptimierende Recommender Systeme
- domänenspezifische Modellgenerierung
- automatisierte Modellwartung
- Gunnar Schröder (gunnar.schroeder@tu-dresden.de)

Algorithmen Integration in Datenbanken
- statistische Programmiersprache R
- Integration in SAP Hana
- Phillip Grosse (philipp.grosse@sap.com)
Schema Flexibility & Graph Data

Data with versatile and changing schema

Flexible Data

Massively and variably interlinked data

Systems
Forecasting in DBMS

Native integration of model-based forecasting inside a DBMS
Optimization of ad-hoc and recurring forecast queries (accuracy + runtime)
Physical design of forecast models in hierarchical data sets

Balancing of energy supply and demand by using flexibilities in time and amount
Forecasting is a fundamental precondition for dynamic scheduling
Energy-domain-specific model creation, model usage and model maintenance
- DB Operator programming in DBL
- Push application logic into Database
- Easy to implement

- SMIX!
- Adaptive access path
- Self-managing indexes
- No explicit index creation anymore

- Darwin Project
- Run multiple QEPs and let the fittest survive
- Compensate bad estimations at runtime
Modern Hardware

- **DEXTER Project / EAST@HAEC**
  - Parallel in-memory indexing
  - Novel query processing style
  - Operational/Live BI
  - SFB 912: HAEC

- **GPU supported query processing**
  - Parallel adaptive QEPs
  - Shrubby Trees
  - GPU programming

- **Parallel Database Primitives**
  - SIMD instructions
  - APU programming
  - Customized CPUs!
  - Distributed processing
Organisatorisches

- Themenauswahl bis nächste Woche
  - LV nur anrechenbar, wenn ein Thema bearbeitet wird

- Nächstes Treffen: Mi, 14.12 @13:00 INF/E009
  - Treffen mit zuständigem Betreuer

- Einarbeitung in das Thema (Literatur, Diskussion, ...)
- Nähere Eingrenzung des zu bearbeitendem Themas
- Bearbeitung des Themas