Hauptspeicher-Datenbanksysteme

- **Hardware-Entwicklungen**
- **Anwendungsstudie: Handelsunternehmen**
- **Column- versus Row-Store**
- **OLAP&OLTP: Snapshotting**
- **Kompaktifizierung**
- **Mehrbenutzersynchronisation**
- **Hochverfügbarkeit**
- **Indexierung**
- **Multi-Core Anfragebearbeitung**

1

Hauptspeicher-Datenbanksysteme

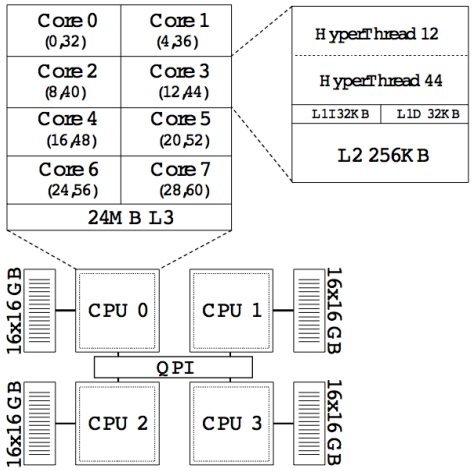
Disk is Tape, Tape is dead … Jim Gray

Die Zeit ist reif für ein Re-engineering der Datenbanksysteme

Man kann heute für 25000 Euro einen Datenbankserver mit 1 TeraByte Hauptspeicher und 32 Rechenkernen kaufen

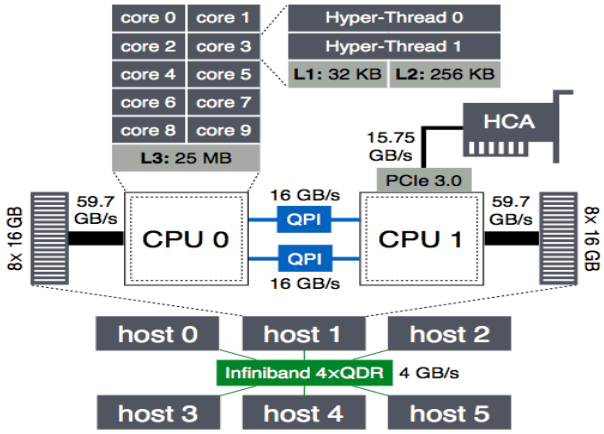
2

Typical NUMA Multi-Core Server with 1 TB DRAM



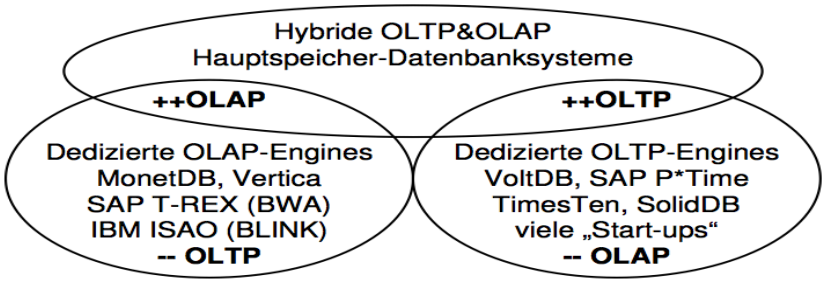
3

Network in the small and in the large



4

Einsatz von Hauptspeicher-Datenbanksystemen



5

Machbarkeits“studie“: Main Memory DBMS

Amazon

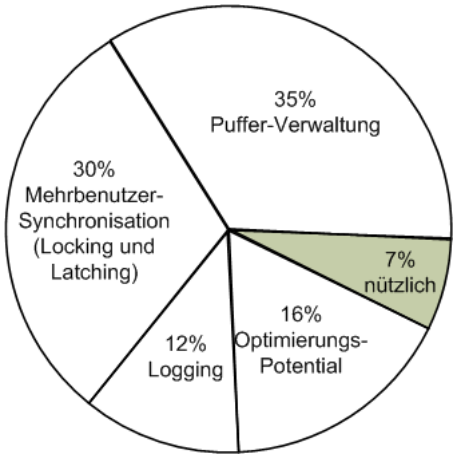
- Data Volume
  - Revenue: 15 billion Euro
  - Avg. Item Price: 15 Euro
  - 1 billion order lines per year
    - 54 Bytes per order line
    - 54 GB per year
    - + additional data
    - - compression
- Transaction Rate
  - Avg: 32 orders per s
  - Peak rate: Thousands/s
  - + inquiries

Intel

- Tera Scale Initiative
- Server with several TB main memory
- We just ordered one from Dell for 49 K Euro
- Main Memory capacity will grow faster than Customers‘ Needs
- Cf. RAMcloud-project at Stanford
  - Ousterhoud et al.

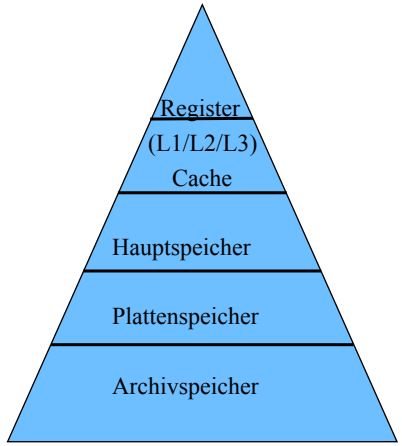
6

Leistungsengpässe: Profiling eines klassischen Datenbanksystems



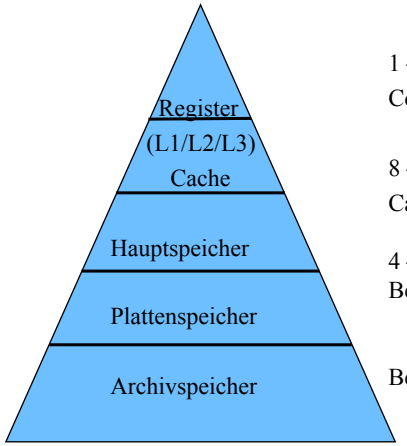
7

Widerholung: Speicherhierarchie



8

Überblick: Speicherhierarchie



1 – 8 Byte
Compiler

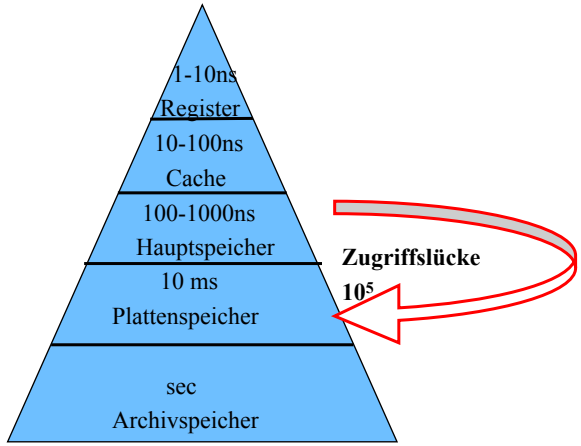
8 – 128 Byte
Cache-Controller

4 – 64 KB
Betriebssystem

Benutzer

9

Überblick: Speicherhierarchie



10

Überblick: Speicherhierarchie

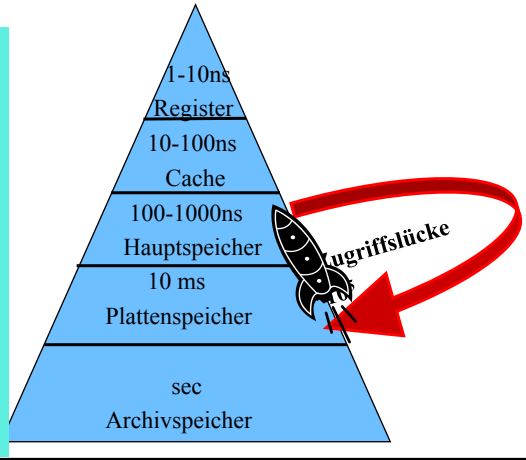
Kopf (1min)

Raum (10 min)

München (1.5h)

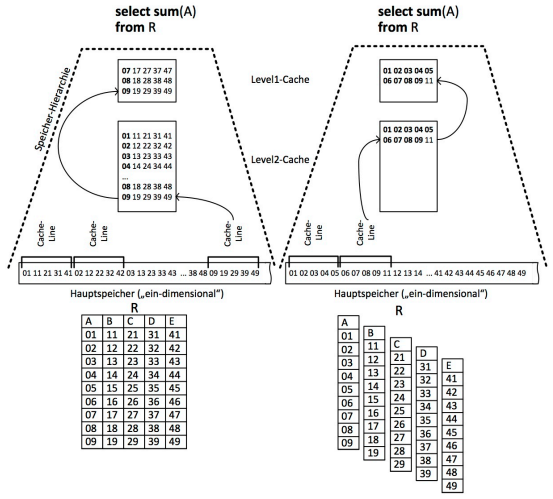
Pluto (2 Jahre)

Andromeda
(2000 Jahre)



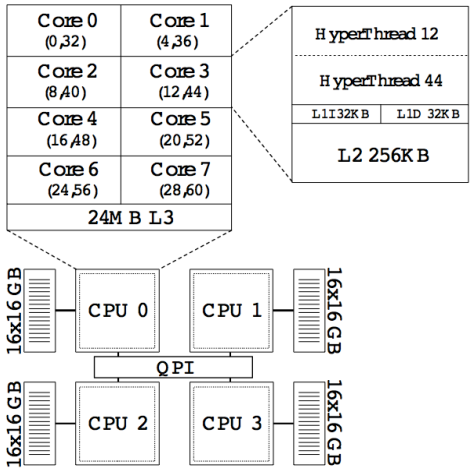
11

Row- versus Column-Store



12

Architektur eines Mehrkernrechners mit NUMA Speicher



Row Store versus Column Store

Verkäufe				
Produkt	Kunde	Preis	Filiale	...
Handy	Kemper	345	Schwabing	...
Radio	Mickey	123	Bogenhausen	...
Handy	Minnie	233	Schwabing	...
Kühlschrank	Urmel	240	Augsburg	...
Beamer	Bond	740	London	...
Handy	Lucie	321	Bogenhausen	...

Row Store versus Column Store

Produkt	
ID	Produkt
0	Handy
1	Radio
2	Handy
3	Kühlschrank
4	Beamer
5	Handy

Kunde	
ID	Kunde
0	Kemper
1	Mickey
2	Minnie
3	Urmel
4	Bond
5	Lucie

Preis	
ID	Preis
0	345
1	123
2	233
3	240
4	740
5	321

Filiale	
ID	Filiale
0	Schwabing
1	Bogenhausen
2	Schwabing
3	Augsburg
4	London
5	Bogenhausen

Anfragebearbeitung

select sum(Preis)
from Verkäufe
where Produkt = 'Handy'

Die schrittweise Abarbeitung dieser Anfrage ist nachfolgend illustriert:

Produkt	
ID	Produkt
0	Handy
1	Radio
2	Handy
3	Kühlschrank
4	Beamer
5	Handy

→ σ_{Handy}

ID
0
2
5

⋈

Preis	
ID	Preis
0	345
1	123
2	233
3	240
4	740
5	321

→ Π_{Preis}

Preis
345
233
321

→ **sum** → 899

Komprimierung

Dictionary	
ID	Wort
0	Augsburg
1	Beamer
2	Bogenhausen
3	Handy
4	Kühlschrank
5	London
6	Radio
7	Schwabing
...	...

Produkt	
ID	Produkt
0	3
1	6
2	3
3	4
4	1
5	3

Filiale	
ID	Filiale
0	7
1	2
2	7
3	0
4	5
5	2

17

Datenstrukturen einer Hauptspeicher-Datenbank

- Kunden: {[id: int, name: char(30), rabatt: double, land: int]}
- Laender: {[id: int, name: char(30), steuern: double]}
- Produkte: {[id: int, name: char(30), preis: double]}
- Verkaeufe: {[id: int, kunde: int, produkt: int, datum: int, preis: double]}

```
create table Kunden
   ( id int, name char(30), rabatt double, land int )
```

18

Row-Store-Format

```
/// Ein Kunde
struct Kunde { unsigned id; char name[30]; double rabatt; unsigned land; };
/// Ein Land
struct Land { unsigned id; char name[30]; double steuern; };
/// Ein Produkt
struct Produkt { unsigned id; char name[30]; double preis; };
/// Ein Verkauf
struct Verkauf { unsigned id; unsigned kunde; unsigned produkt;
                 unsigned datum; double preis; };

/// Die Datenbank im Row-Format
vector<Kunde> Kunden;
vector<Land> Laender;
vector<Produkt> Produkte;
vector<Verkauf> Verkaeufe;
```

19

Column-Store-Format

```
/// Template für Strings fester Länge -- ohne Indirektion
template <unsigned len> struct Char { char data[len]; };

/// Ein Kunde
struct Kunde { unsigned id; Char<30> name; double rabatt; unsigned land; };
///Alle Kunden in Column-Format
struct Kunden {
   vector<unsigned> data_id; vector<Char<30>> data_name;
   vector<double> data_rabatt; vector<unsigned> data_land;

   void insert(Kunde&& kunde);
};

/// Ein Land
struct Land { unsigned id; char name[30]; double steuern; };
/// Alle Länder in Column-Format
struct Laender {
   vector<unsigned> data_id; vector<Char<30>> data_name;
   vector<double> data_steuern;

   void insert(Land&& land);
};
```

20

Column-Store-Format (cont'd)

```
/// Ein Produkt
struct Produkt { unsigned id; char name[30]; double preis; };
/// Alle Produkte in Column-Format
struct Produkte {
   vector<unsigned> data_id; vector<Char<30>> data_name;
   vector<double> data_preis;

   void insert(Produkt&& produkt);
};

/// Ein Verkauf
struct Verkauf { unsigned id; unsigned kunde; unsigned produkt;
                 unsigned datum; double preis; };
/// Alle Verkäufe in Column-Format
struct Verkaeufe {
   vector<unsigned> data_id; vector<unsigned> data_kunde;
   vector<unsigned> data_produkt; vector<unsigned> data_datum;
   vector<double> data_preis;

   void insert(Verkauf&& verkauf);
};
```

21

Einfügeoperation eines Tupels

Insert into Verkaeufe values (12, 007, 4711, 27.50)

```
void Verkaeufe::insert(Verkauf&& verkauf)
{
   data_id.push_back(verkauf.id);
   data_kunde.push_back(verkauf.kunde);
   data_produkt.push_back(verkauf.produkt);
   data_datum.push_back(verkauf.datum);
   data_preis.push_back(verkauf.preis);
}
```

22

Anfragen

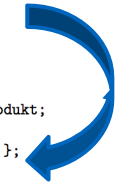
```
select sum(v.preis) from Verkaeufe v
where v.datum >= 20130101
```

```
double umsatz(Verkaeufe& v)
{
   double summe = 0.0;
   for (unsigned i = 0; i < v.data_datum.size(); i++) {
      if (v.data_datum[i] >= 20130101) {
         summe += v.data_preis[i];
      }
   }
return summe;
}
```

23

Hybrides Speichermodell

```
select datum, sum(preis)
from Verkaeufe
where datum >= 20130101
group by datum

/// Ein Verkauf
struct Verkauf { unsigned id; unsigned kunde; unsigned produkt;
                 unsigned datum; double preis; };
struct VerkaufsDatumPreis { unsigned datum; double preis; };
/// Alle Verkäufe in hybridem Format
struct Verkaeufe {
 vector<unsigned> data_id; vector<unsigned> data_kunde;
 vector<unsigned> data_produkt;
 vector<VerkaufsDatumPreis> data_datum_preis;

 void insert(Verkauf&& verkauf);
};
```



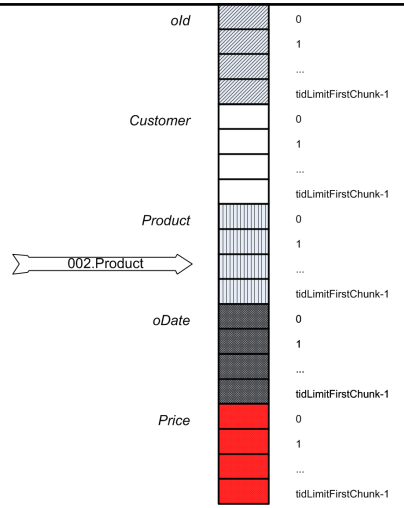
24

Anfragebearbeitung

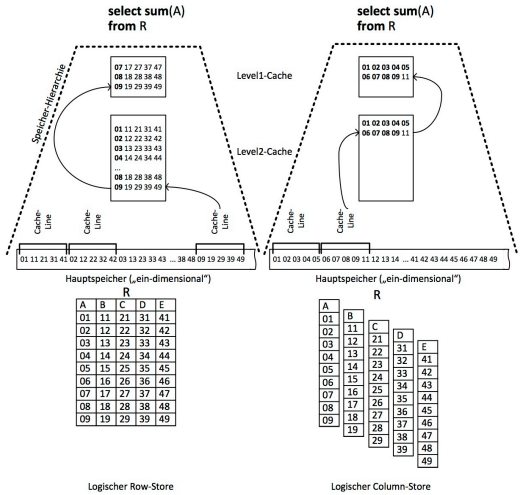
```
unordered_map<unsigned, double> umsatzProDatum(Verkaeufe& verkaeufe)
{
   unordered_map<unsigned, double> groupBy;
   for (VerkaufsDatumPreis datum_preis : verkaeufe.data_datum_preis) {
      if (datum_preis.datum >= 20130101) {
         groupBy[datum_preis.datum] += datum_preis.preis;
      }
   }
   return groupBy;
}
```


HyPer ist ein Column Store

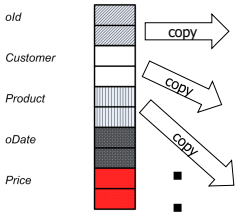
create table myOrders (
old **int**,
Customer **int**,
Product **int**,
oDate **date**,
Price **decimal(10,2)**)



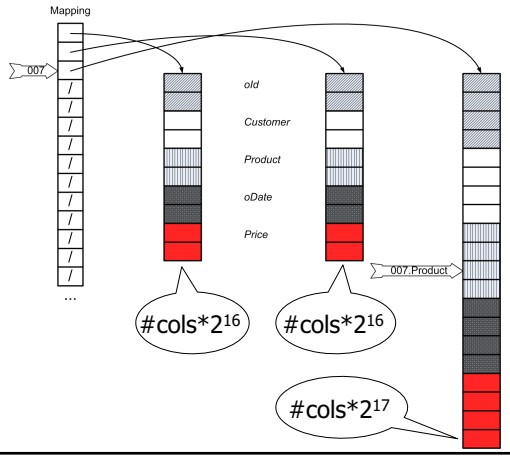
Warum Column-Store?



Dynamic Growth: Initially Doubling and Copying from 2^4 to 2^5 up to 2^{16}



Beyond 2^{16}: Data Partitioning and Dynamic Growth



Data Partitioning and Dynamic Growth

```
// Parts of hyper/rts/runtime/DataPartition.cpp

DataPartition::RowAccess::RowAccess(DataPartition& part,uint64_t tid)
   // Constructor
{
   auto& desc=*part.description;
   if (tid<tidLimitFirstChunk) {
      // ...
   } else {
      unsigned ctlz=bitops<uint64_t>::clz(tid);
      unsigned chunkId = (64-DataPartition::maxFirstChunkBits)-ctlz;
      unsigned chunkCapacityInBits=(64-1)-ctlz;
      auto chunk=part.chunks[chunkId];
      uint64_t chunkOffset=(1ull<<chunkCapacityInBits);
      uint64_t chunkIndex=tid-chunkOffset;
      // ...
   }
}
```

2^{16}

Efficient address Transformation

Special Treatment of Strings:

→*RuntimeString*

Store short Strings inside the Data Vectors
- Length & actual string of length up to 12 bytes

For Long Strings the Data Vector contains
- Length of the string
- 4 Byte Prefix of the string (→short-cut for comparisons)
- Pointer to the String

Thus, strings always have fixed 16 byte representation in the Data Vector

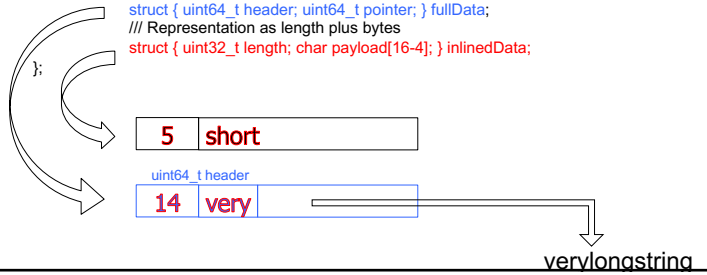
```
/// The representation:
/// length (4 bytes), prefix (4 bytes, padded with null)
/// (pointer to string if length>12) or (rest of the string if length <=12)
union {
        /// Representation as  header+pointer
        struct { uint64_t header; uint64_t pointer; } fullData;
        /// Representation as length plus bytes
        struct { uint32_t length; char payload[16-4]; } inlinedData;
};
```


Special Treatment of Strings:

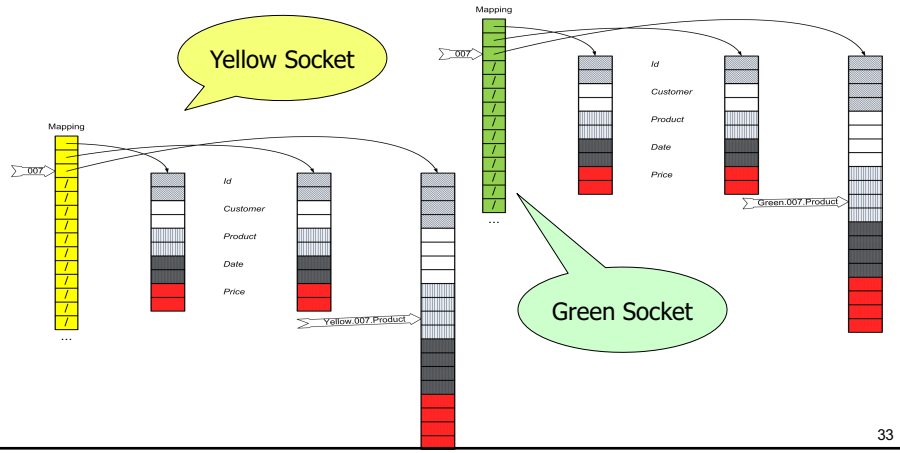
→*RuntimeString*

Thus, strings always have fixed 16 byte representation in the Data Vector

```
/// The representation:
/// length (4 bytes), prefix (4 bytes, padded with null)
/// (pointer to string if length>12) or (rest of the string if length <=12)
union {
        /// Representation as header+pointer
        struct { uint64_t header; uint64_t pointer; } fullData;
        /// Representation as length plus bytes
        struct { uint32_t length; char payload[16-4]; } inlinedData;
};
```

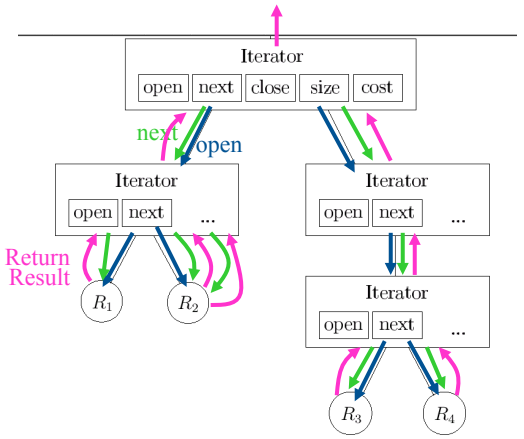


NUMA (Non Uniform Memory Architecture) Support: Partitioning



33

Traditional Pull-based (Volcano-Iterator-based) Query Processing



34

HyPer`s Data-Centric Code Generation

```
select *
from R, S, T
where T.x=7 and S.y=3 andD R.z>5 and
      T.B=S.B and S.A=R.A
```

$$\sigma_{z>5}R \bowtie_A \sigma_{y=3}S \bowtie_B \sigma_{x=7}T$$

35

Pipelines are compiled into holistic data-centric LLVM code fragments

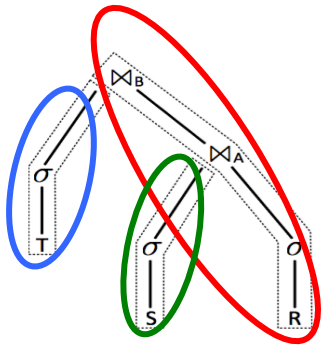
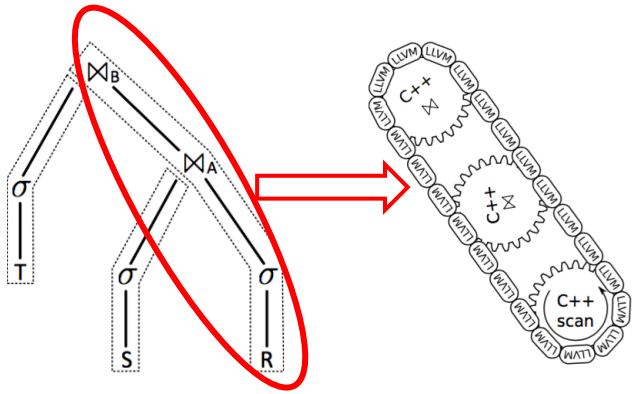


Figure 4.10: Algebra tree

initialize memory of $\bowtie_A$, $\bowtie_B$
for each tuple t in T
  if $t.x = 7$
    materialize t in hash table of $\bowtie_B$
for each tuple s in S
  if $s.y = 3$
    materialize s in hash table of $\bowtie_A$
for each tuple r in R
  if $r.z > 5$
    for each match s in $\bowtie_B$ $[r.B]$
      for each match t in $\bowtie_A$ $[s.A]$
        output $r \circ s \circ t$

36

Not all code is dynamically generated: Interplay of pre-defined C++ code and dynamically generated LLVM



37

Abstraction of the Query Compiler: Produce/Consume

```
scan.produce():
    print "for each tuple in relation"
    scan.parent.consume(attributes,scan)
σ.produce:
    σ.input.produce()
σ.consume(a,s):
    print "if "+σ.condition
    σ.parent.consume(attr,σ)
```

- produce()
  - generates code to compute the result tuples of the operator
- consume(attrs,source)
  - Generates code for processing one tuple

```
⋈.produce():
    ⋈.left.produce()
    ⋈.right.produce()
⋈.consume(a,s):
    if (s==⋈.left)
        print "materialize tuple in hash table"
    else
        print "for each match in hashtable[" +a.joinattr+"]"
        ⋈.parent.consume(a+new attributes)
```

- Compile time operations
  - Not called at run time
- Simple interface (just like the iterator model)
  - However code generation functions; not processing functions

38

Pipelines are compiled into holistic data-centric LLVM code fragments

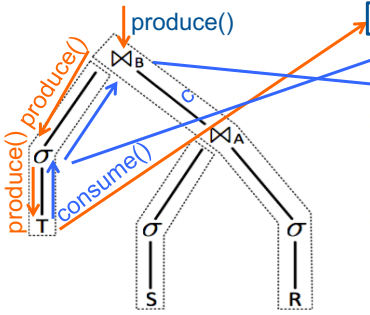


Figure 4.10: Algebra tree

```
initialize memory of ⋈A, ⋈B
for each tuple t in T
    if t.x = 7
        materialize t in hash table of ⋈B
for each tuple s in S
    if s.y = 3
        materialize s in hash table of ⋈A
for each tuple r in R
    if r.z > 5
        for each match s in ⋈B [r.B]
            for each match t in ⋈A [s.A]
                output r ∘ s ∘ t
```

39

Generating efficient code for modern Processors

The pipeline code that is generated by the
- **consume()** and
- **produce()**

is machine-agnostic (i.e., is independent of the processor on which it is executed)

This is the benefit of using the machine-independent compiler framework LLVM (Low Level Virtual Machine)

LLVM optimizes the generated code for the particular target machine

However, this does not optimize the execution for
- **massively parallel execution** on multi-core machines
- **local processing on NUMA** (non-uniform memory access) machines where DRAM is segmented into multiple regions

This is built into the runtime execution system of HyPer

40

Anwendungsoperationen in der Datenbank: Stored Procedures

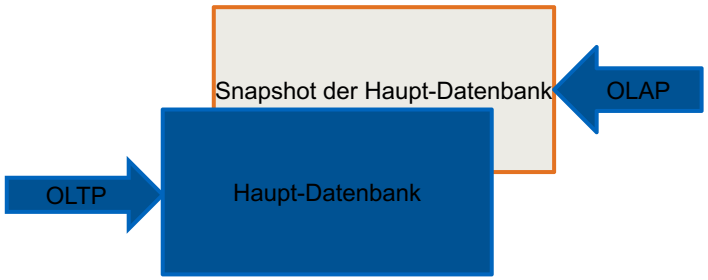
```
create procedure newOrder (w_id integer not null, d_id integer not null,
      c_id integer not null, table positions(line_number integer not null,
      supware integer not null, itemid integer not null, qty integer not null),
      datetime timestamp not null)  // note the TABLE-valued parameter above
{
   select w_tax from warehouse w where w.w_id=w_id; // w_tax value used later
   select c_discount from customer c // c_discount used in orderline insert
       where c_w_id=w_id and c_d_id=d_id and c.c_id=c_id;
```

41

```
   select d_next_o_id as o_id,d_tax from district d // get the next o_id
       where d_w_id=w_id and d.d_id=d_id;
   update district set d_next_o_id=o_id+1 // increment the next o_id
       where d_w_id=w_id and district.d_id=d_id;

   select count(*) as cnt from positions; // how many items are ordered
   select case when count(*)=0 then 1 else 0 end as all_local
       from positions where supware<>w_id;

   insert into "order" values (o_id,d_id,w_id,c_id,datetime,0,cnt,all_local);
   insert into neworder values (o_id,d_id,w_id); // insert reference to order

   update stock
   set s_quantity=case when s_quantity>qty then s_quantity-qty
                                           else s_quantity+91-qty end,
       s_remote_cnt=s_remote_cnt+case when supware<>w_id then 1 else 0 end,
       s_order_cnt=s_order_cnt+case when supware=w_id then 1 else 0 end
   from positions
   where s_w_id=supware and s_i_id=itemid;

   insert into orderline // insert all the order positions
      select o_id,d_id,w_id,line_number,itemid,supware,null,qty,
             qty*i_price*(1.0+w_tax+d_tax)*(1.0-c_discount),
             case d_id when 1 then s_dist_01 when 2 then s_dist_02
                      when 3 then s_dist_03 when 4 then s_dist_04
                      when 5 then s_dist_05 when 6 then s_dist_06
                      when 7 then s_dist_07 when 8 then s_dist_08
                      when 9 then s_dist_09 when 10 then s_dist_10 end
      from positions, item, stock
      where itemid=i_id and s_w_id=supware and s_i_id=itemid
   returning count(*) as inserted; // how many were inserted?

   if (inserted<cnt) rollback; // not all ==> invalid item ==> abort
};
```

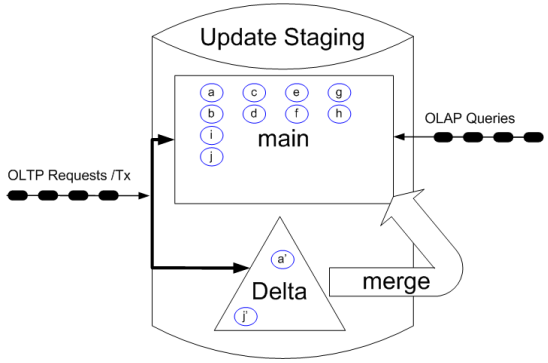
42

Snapshots für Anfragen



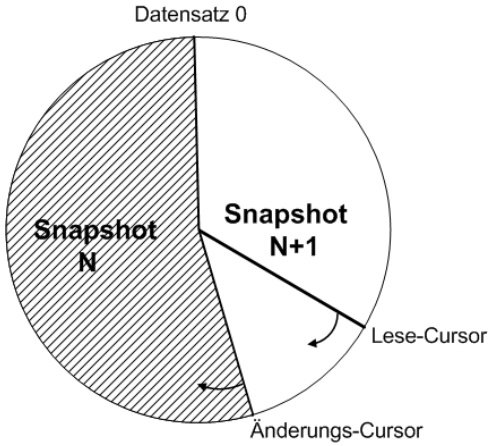
43

Update Staging: In vielen Systemen verwendet, zB. Hana von SAP



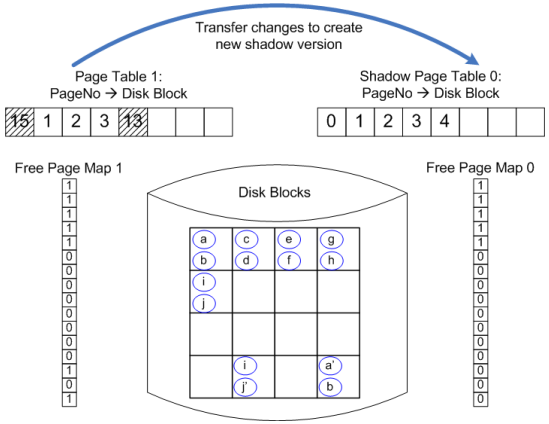
44

Scan-only Datenbanken: ISAO von IBM oder Crescando von der ETHZ

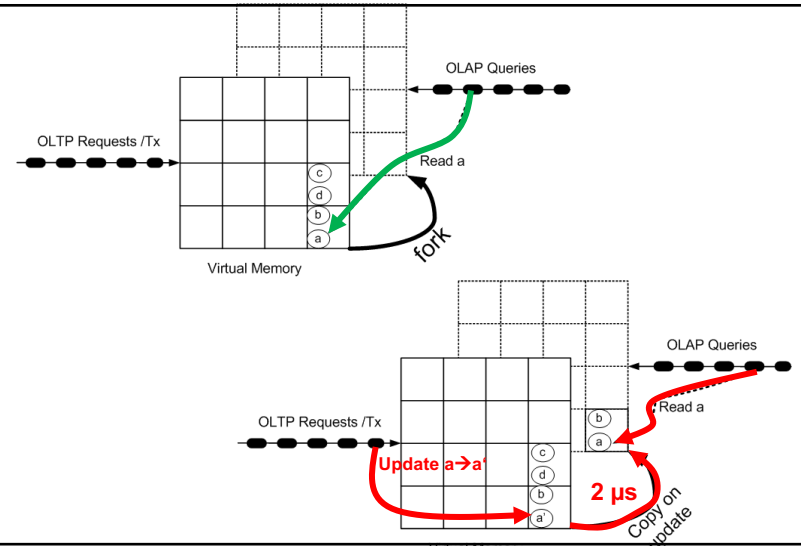


45

Ursprüngliches Schattenspeicher-Verfahren: Lorie77 für IBM System R

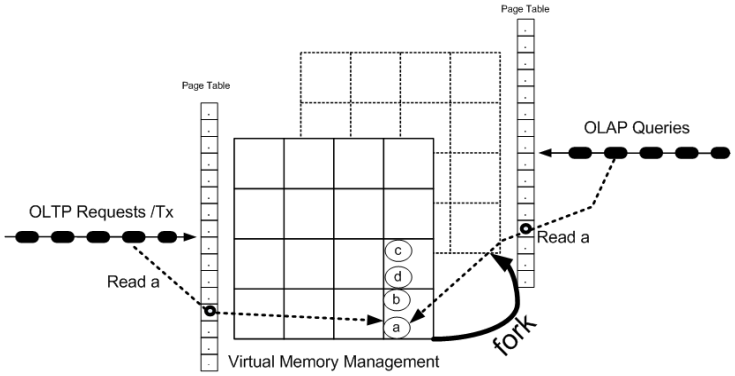


46



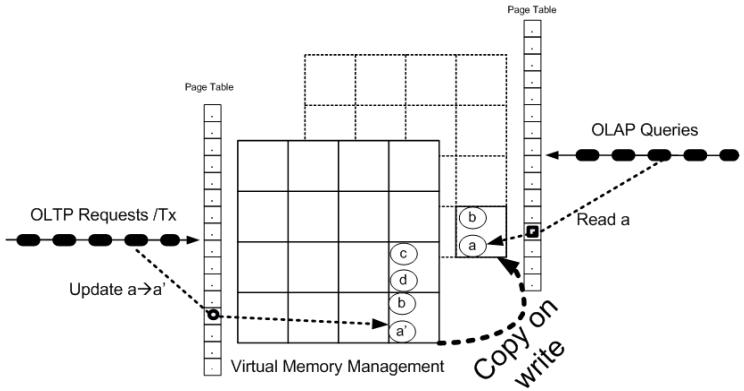
47

Snapshotting via fork-ing: Details



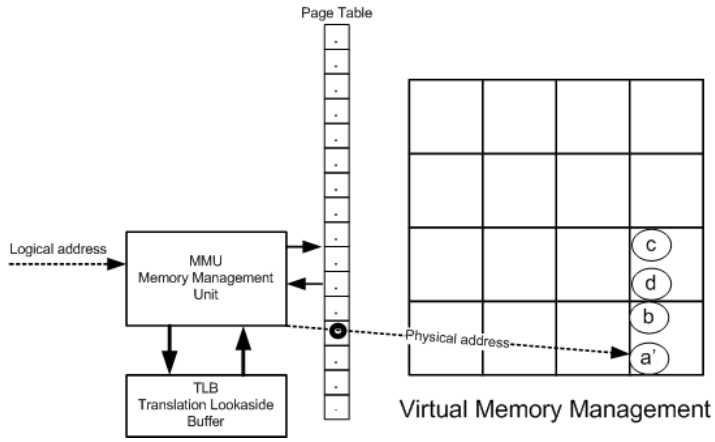
48

Snapshot Maintenance: copy on write



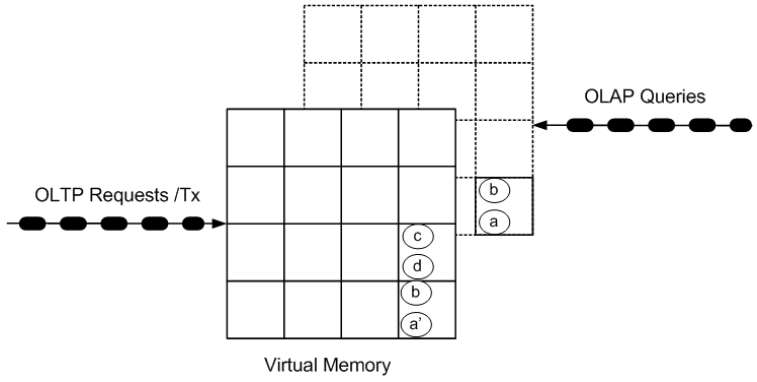
49

Fast because of Hardware-Support: MMU



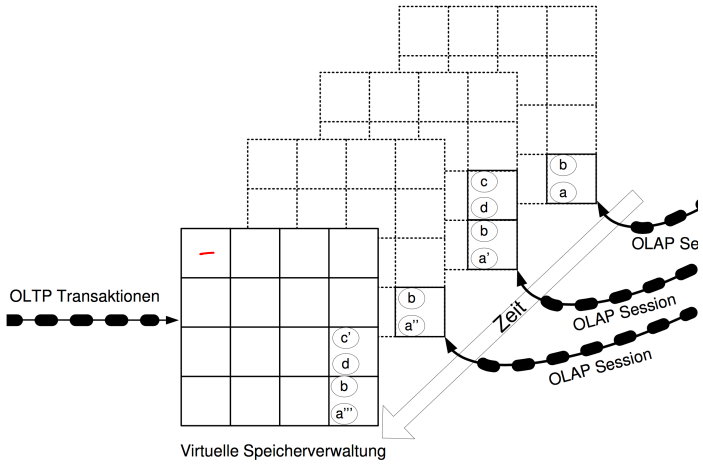
50

OLAP Queries on Tx-Consistent Snapshots



51

Multiple Query Sessions



52

Synchronization-Assertions

Serializability of the OLTP Transactions
- What else if executed serially
- We support full ACID → see coming slides

Snapshot isolation of the OLAP queries
- Multi-version mixed synchronization method
- Several OLAP queries form one Tx = OLAP Session
- Bernstein, Hadzilacos, Goodman: Chapter 5.5

Kompaktifizierung: Motivation

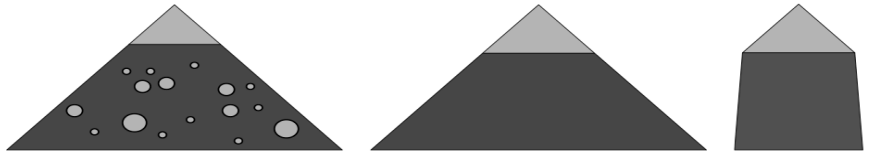
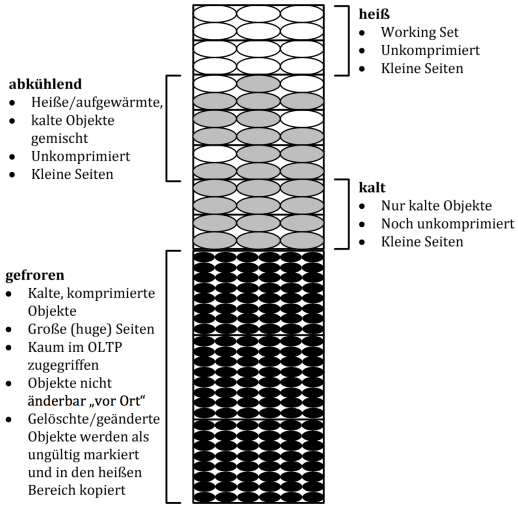
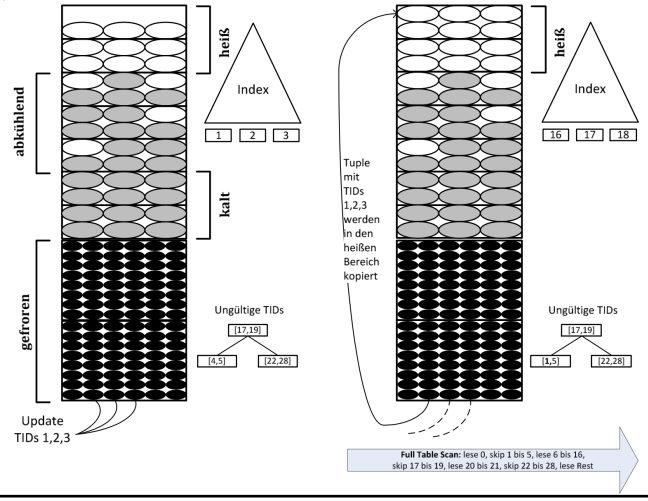


Abbildung 18.12: Der Working Set einer Datenbank: (links) verstreut über die DB, (mittig) nach Reorganisation der heißen Objekte in einen Bereich, (rechts) nach zusätzlicher Kompression der kalten Datenbank

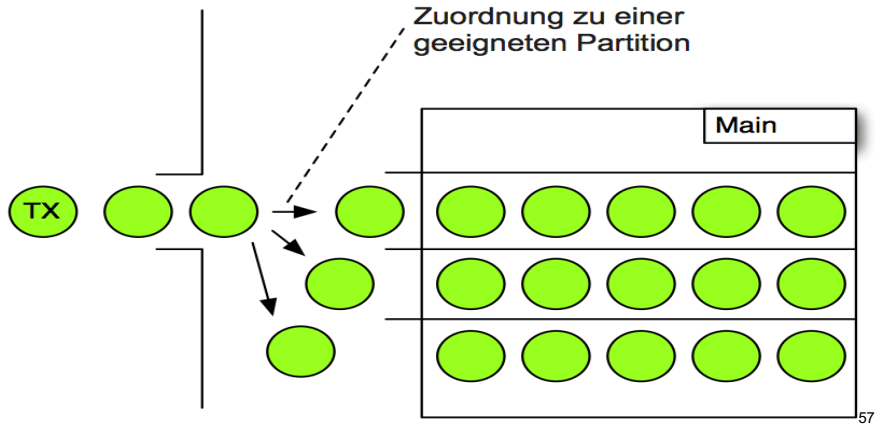
Kompaktifizierung der Datenbank



Invalidierung gefrorener Datenobjekte

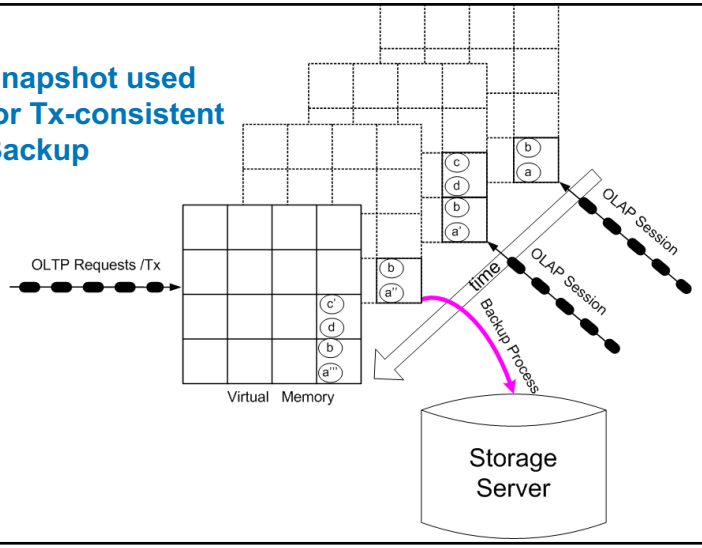


Transaktionsverwaltung: serielle Ausführung auf Partitionen



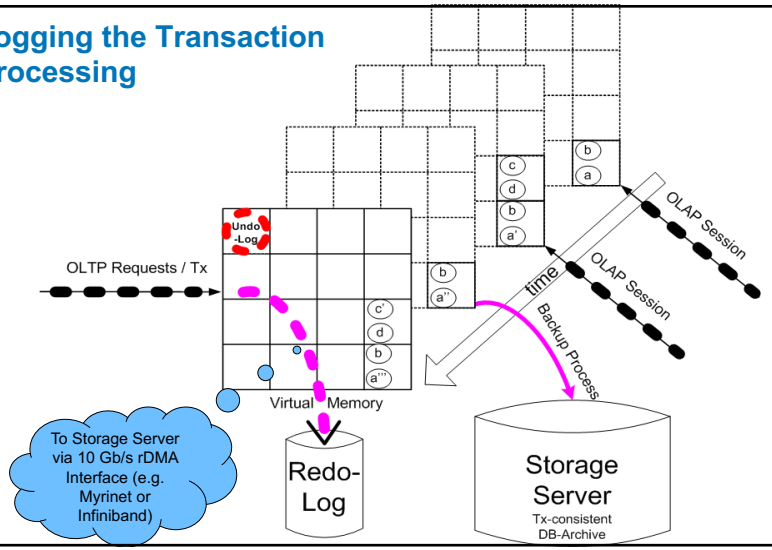
57

Snapshot used for Tx-consistent Backup



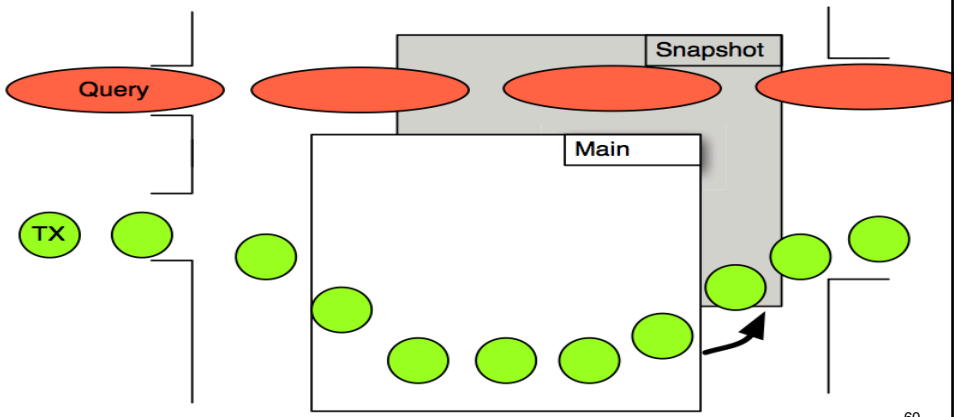
58

Logging the Transaction Processing



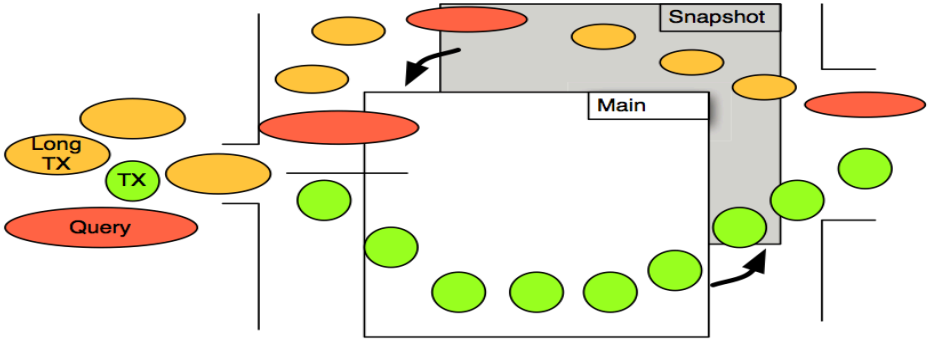
59

Isolation von OLAP und OLTP



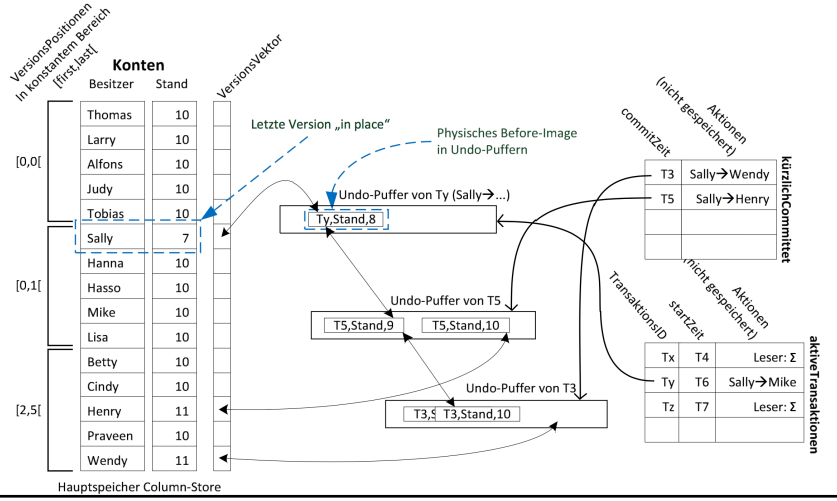
60

Tentative Ausführung langer Transaktionen



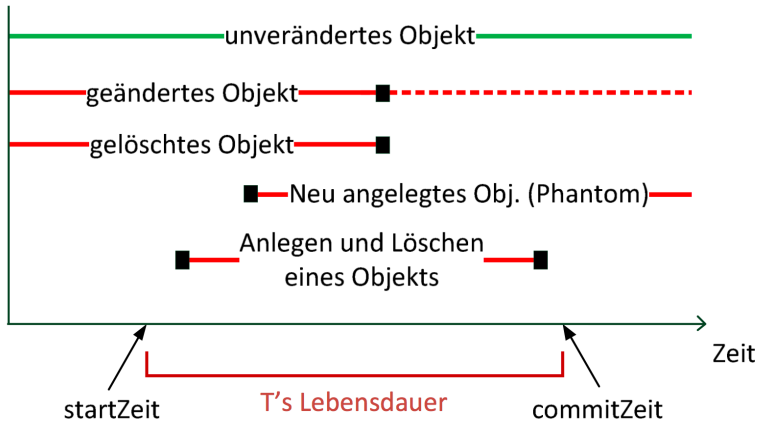
61

MVCC: Multi-Version Concurrencv Control



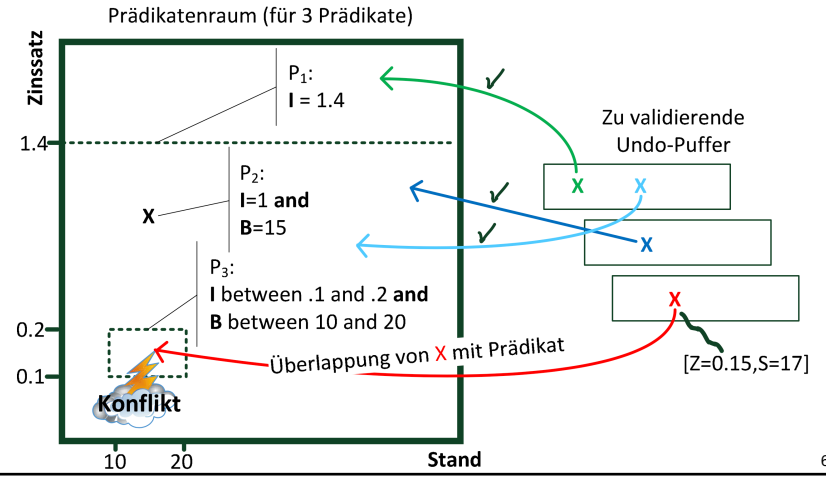
62

Validierung einer Update-Transaktion



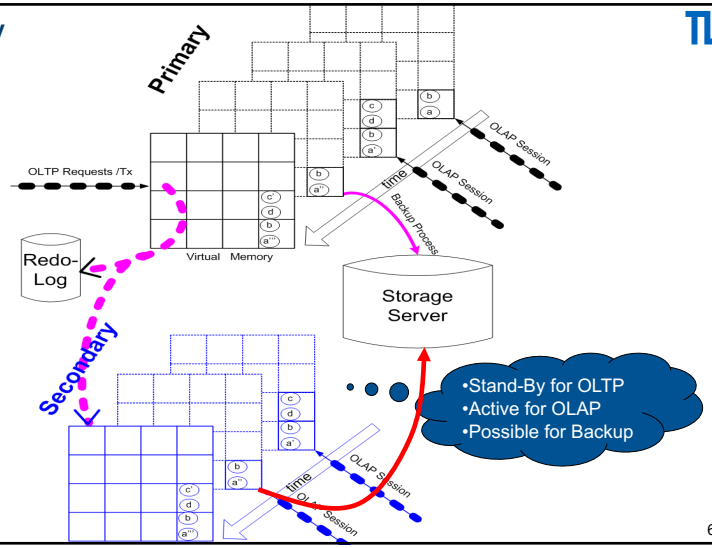
63

Precision Locking: Prädikatprüfung

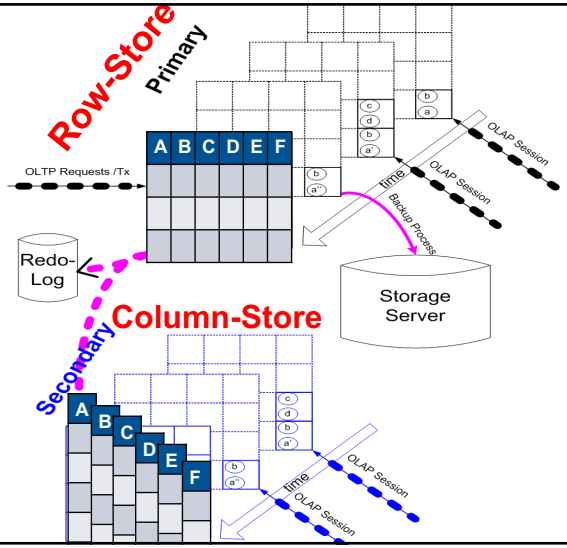


64

High Availability & Load Balancing



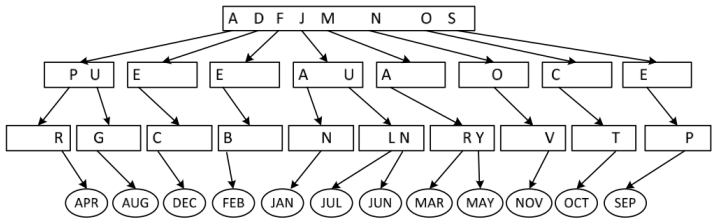
65



66

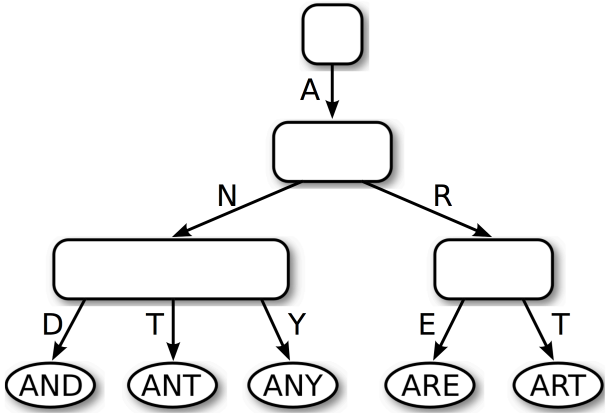
Indexstrukturen für Hauptspeicher-Datenbanken

Radix-Baum / Trie / Präfixbaum



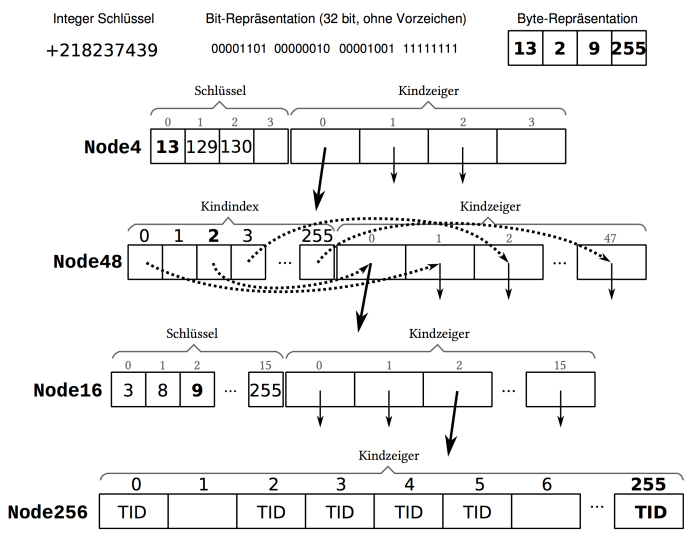
67

Idee des Adaptiven Radix-Baums ART



68

Adaptive Knoten des ART-Baums



69

Join-Berechnung

Cache-Lokalität

Mehrkern-Parallelität

NUMA-Berücksichtigung

Synchronisations-freie Parallelität

70

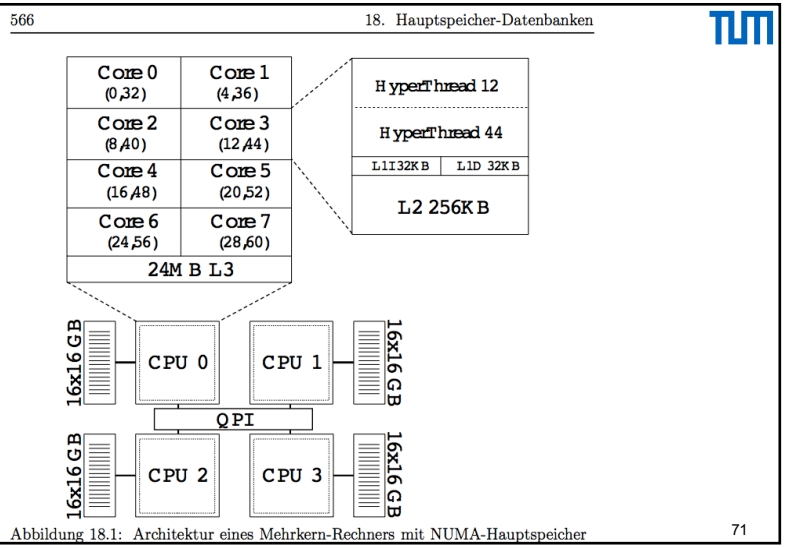
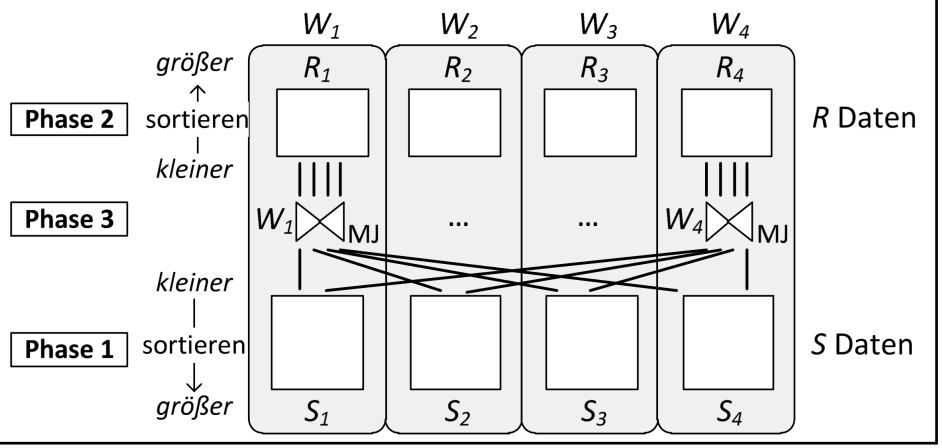


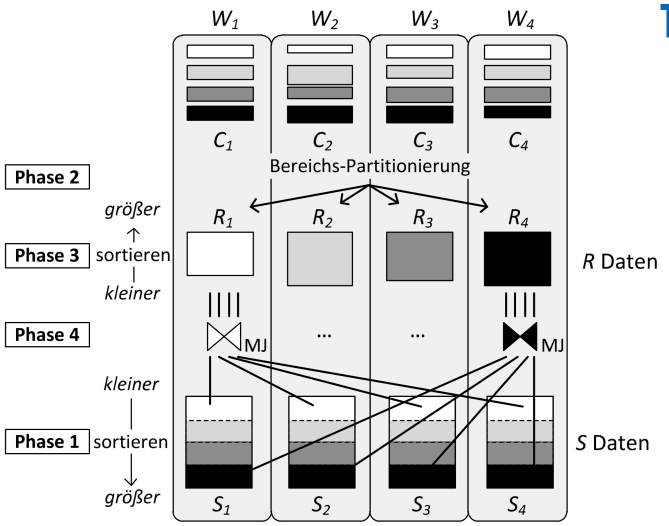
Abbildung 18.1: Architektur eines Mehrkern-Rechners mit NUMA-Hauptspeicher

71

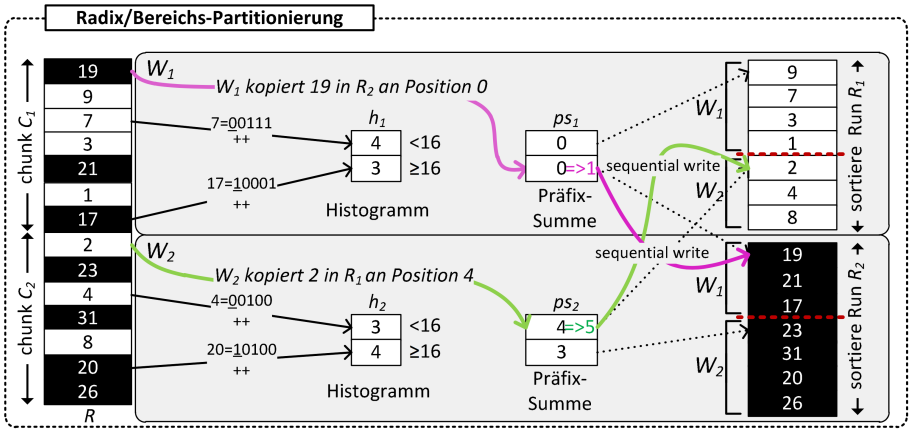
Grundidee des hoch-parallelen Sort/Merge-Joins



Bereichs-Partitionierung

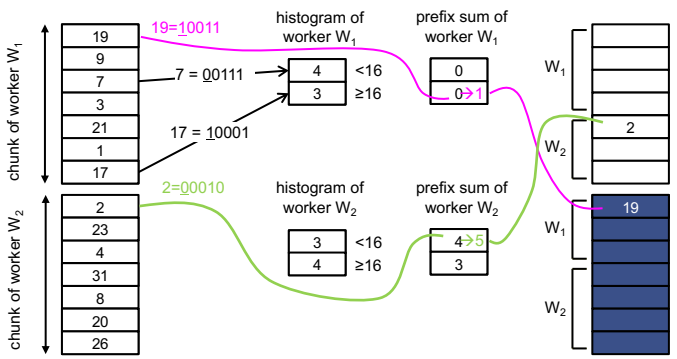


Hochparallel Bereichs/Radix-Partitionierung



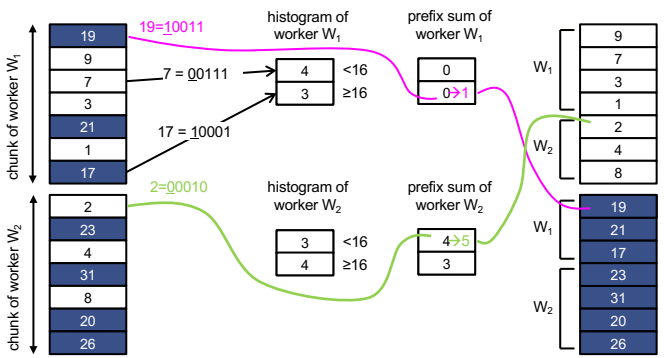
Technische Universität München

Range partitioning of private input



Technische Universität München

Range partitioning of private input



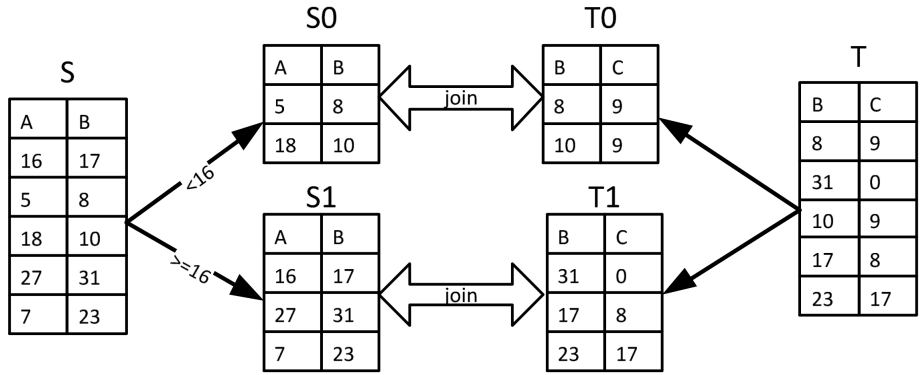
Real C hacker at work …

$$ps_i[j] = \&R_j[(\sum_{k=1}^{i-1} h_k[j])]$$

$$\texttt{memcpy}(ps_i[sp[t.key \gg (64-B)]]\texttt{++}, t, t.size)$$

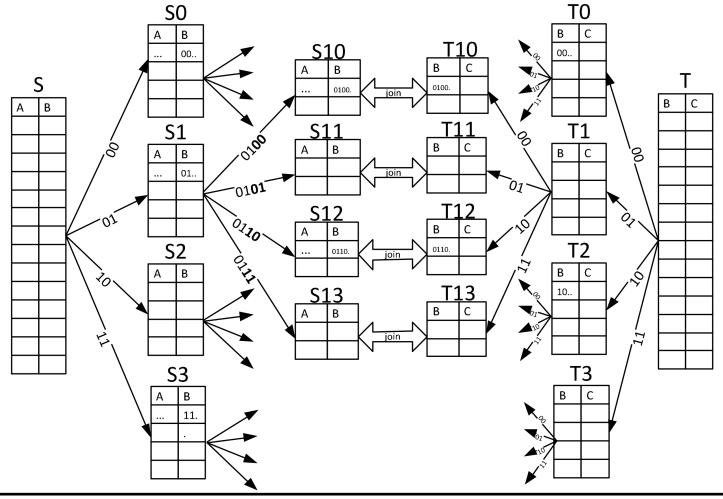
77

Paralleler Radix-Join



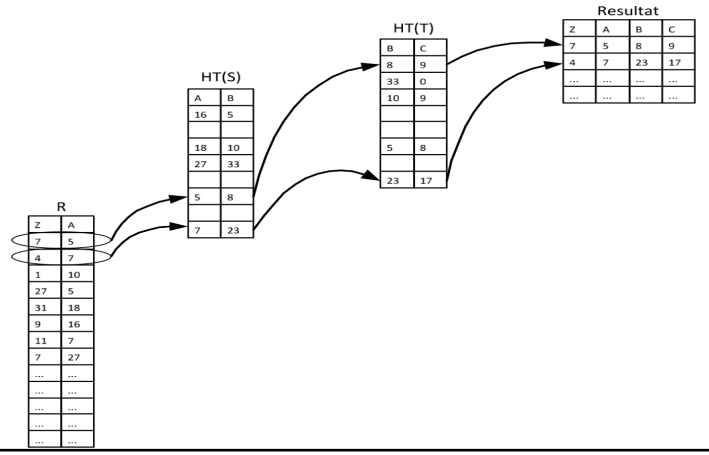
78

Mehrfache Partitionierung des Radix-Joins: Cache-Lokalität



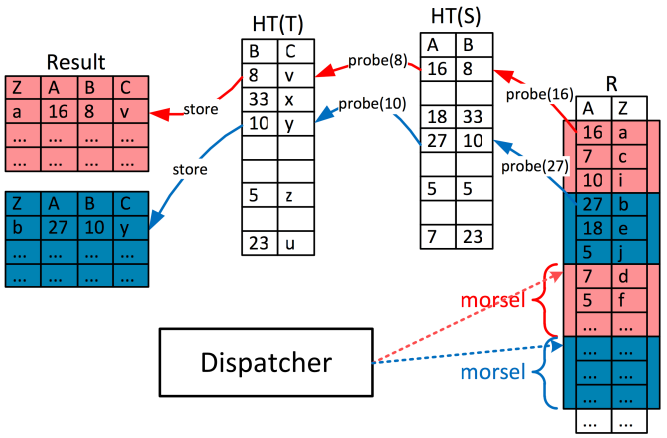
79

Hash-Join-Teams: Globale Hashtabelle



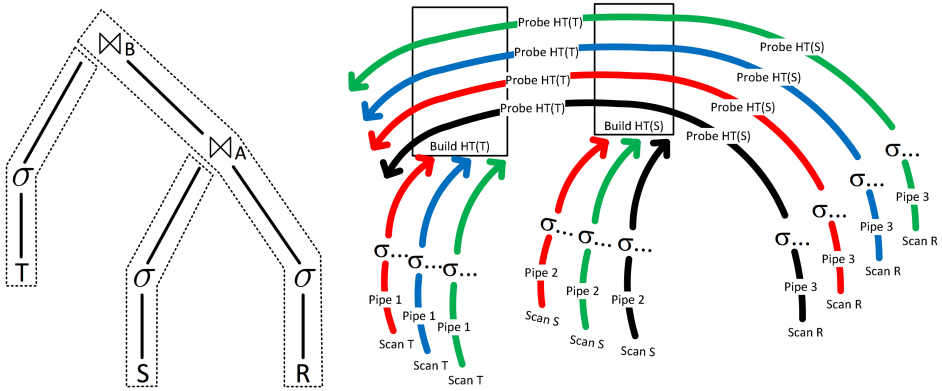
80

NUMA-lokale Arbeitszuteilung:
„Häppchen“-weise (morsel driven)



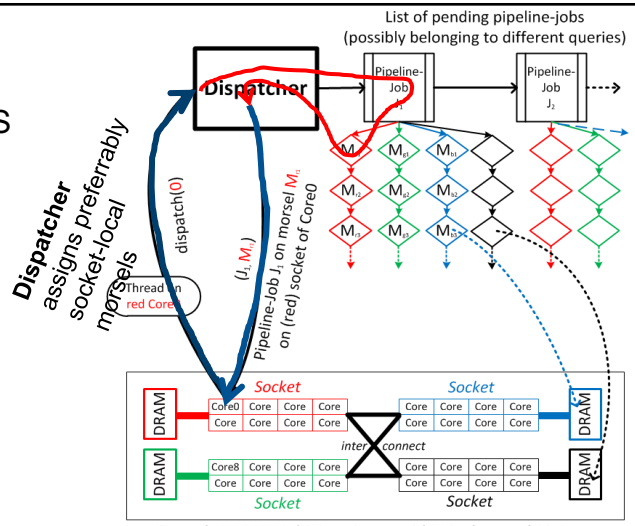
81

Adaptive Dynamische Parallelisierung



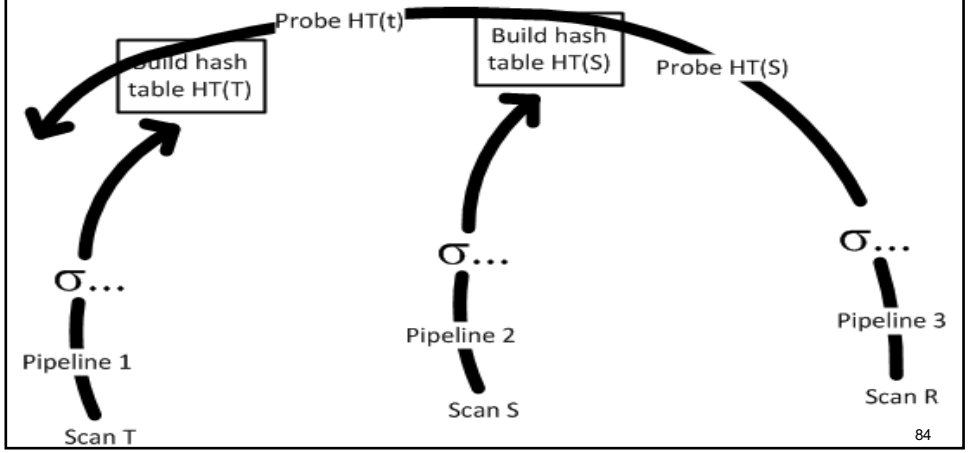
82

Dispatcher:
lock-free DS



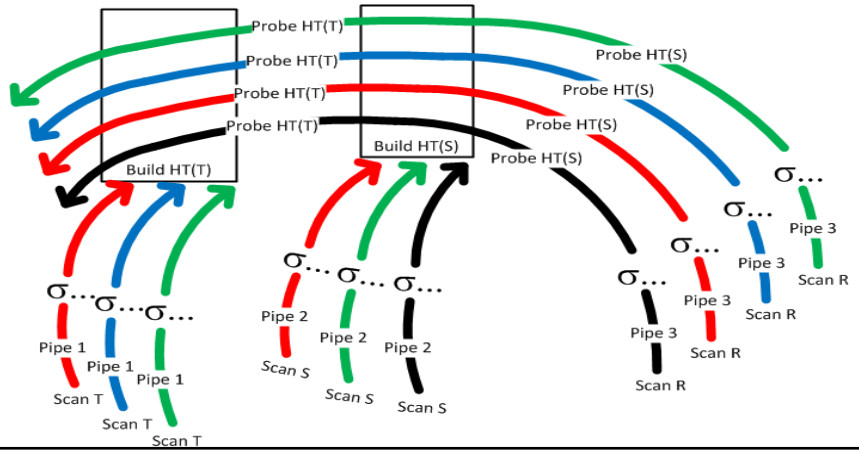
Example NUMA Multi-Core Server with 4 Sockets and 32 Cores

83

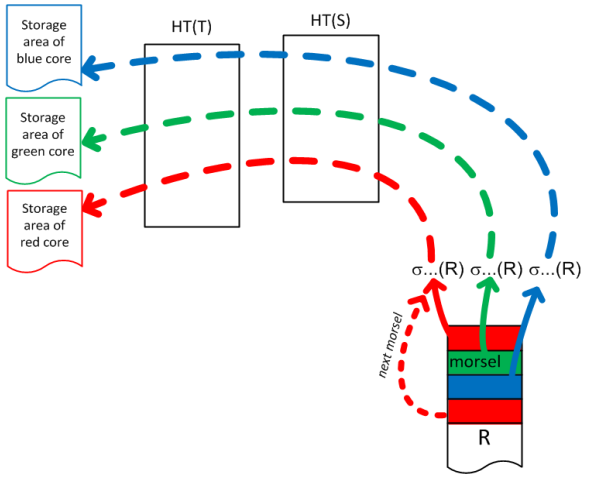


84

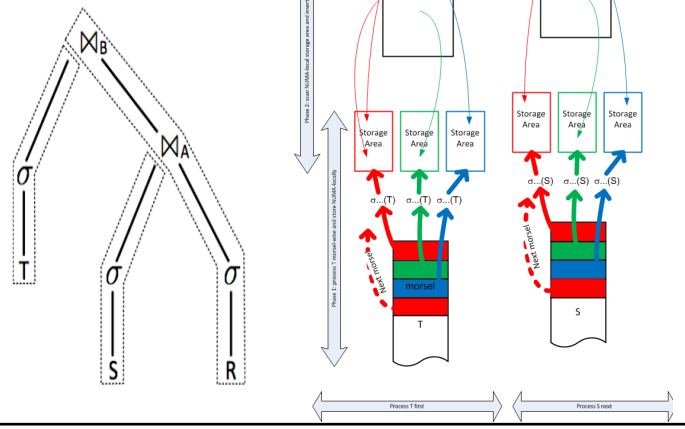
Adaptive Parallelism



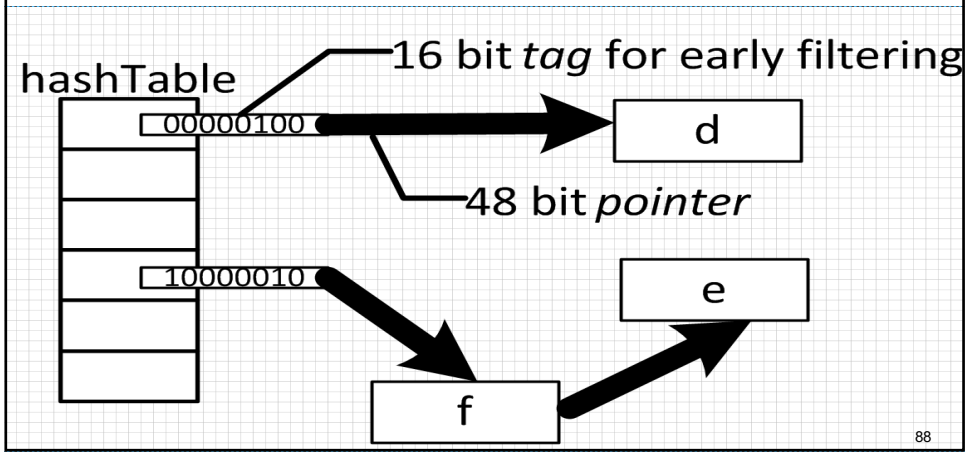
NUMA-local data processing



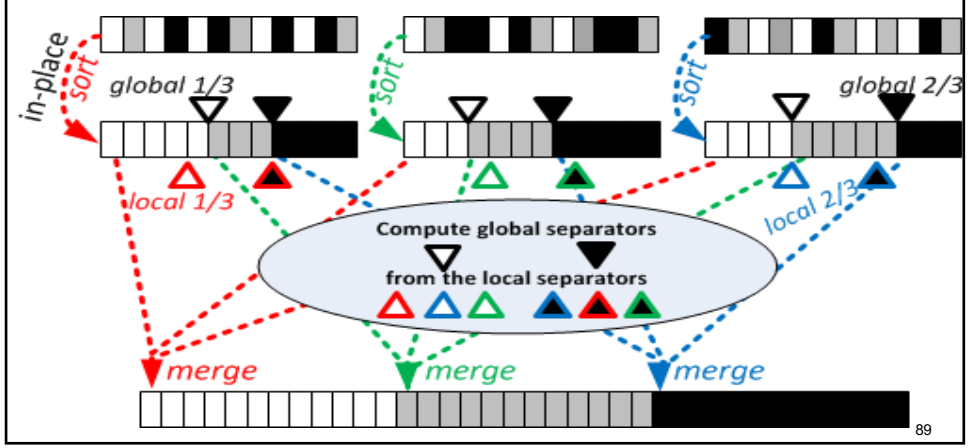
Building the Hash Tables



Hash Table with small Bloom Filters

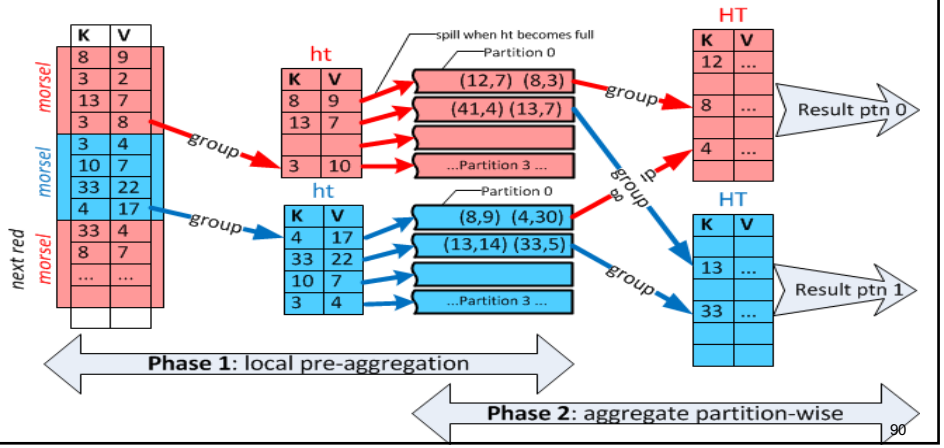


Parallel Sorting: (1) Sort locally (2) Determine Separators (3) Merge



89

Parallel Aggregation: (1) pre-aggregation (2) global grouping



90

Der natürliche Verbund zweier Relationen R und S

R		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

$\bowtie$

S		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

$=$

$R \bowtie S$				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_3	b_3	c_1	d_1	e_1
a_5	b_5	c_3	d_2	e_2

91

Algorithmen auf sehr großen Datenmengen

R ∩ S

R
2
3
44
5
78
90
13
17
42
89

- Nested Loop: $O(N^2)$
- Sortieren: O(N log N)
- Partitionieren und Hashing

S
44
17
97
5
6
27
2
13
9

92