A cost-aware strategy for merging differential stores in column-oriented in-memory DBMS

Florian Hübner[1], Joos-Hendrik Böse[2],
Jens Krüger[1], Frank Renkes[2],
Cafer Tosun[2], Alexander Zeier[1], Hasso Plattner[1]

1 = Hasso-Plattner-Institut, 2 = SAP AG

Outline

- **Preliminaries**
- Value Distrib. in Enterprise Data
- Partial Merge Algorithm
- Evaluation

Motivation and Mission

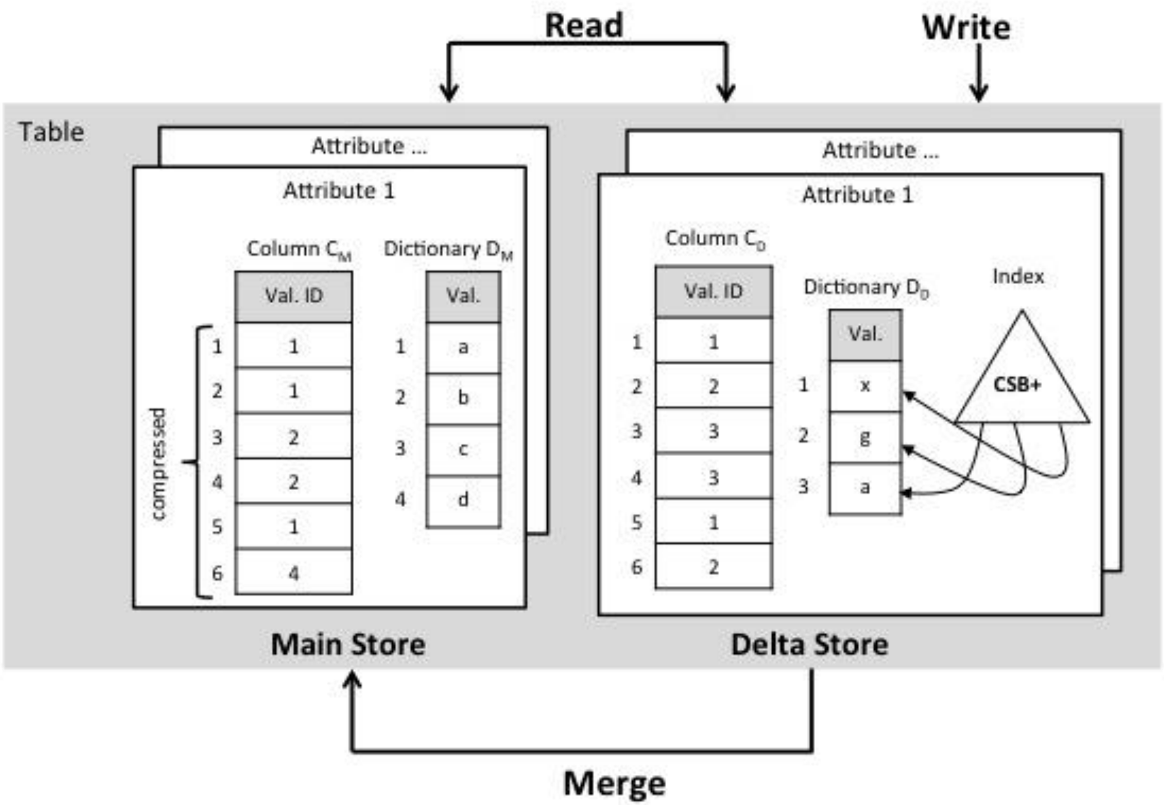
- Context of this work:
  - SAP and HPI are working on consolidation of analytical and transactional workloads in a single column-oriented in-memory DBMS
- Current Setting:
  - Column-oriented DBMSs favor analytical queries, while write operations on read-optimized column structures are expensive
  - Common solution: maintain a write-optimized delta structure in addition to the read-optimized structure (e.g., in C-Store, MonetDB, NewDB, HyRise)
- General Problem:
  - Both structures must be merged for read performance and memory consumption
  - Merging both structures induces substantial write overhead
  - This work contributes on how to optimize this merge process

System Model and Data Structures

- Tables consist of a main store (read-optimized) and delta store (write-optimized)

- Main store:
  - **Sorted dictionary** D_M
  - Column data C_M is a vector of valueIDs
  - valueIDs are defined by value position in D_M
  - Compression

- Delta store:
  - Dictionary D_D is **unsorted**
  - Index (CSB+) is used for valueID lookup in $log(|D_{D|})$
  - No compression but dictionary



Large delta store sizes compromise overall read performance and space consumption

- **Decreasing range select performance**
  - **Range selects** need an additional calculation and indirection in order to be answered for the delta store
  - They can be calculated directly on valueIDs within the main store

- **Less efficient compression of data in delta structure**
  - Run-length encoding, common value suppression, cluster coding, …

- **Increasing size of index for delta dictionary**
  - CSB+-tree needed for fast lookups in unsorted delta dictionary while the main store's sorted dictionary allows for binary search directly -> additional space consumption
  - For large delta dictionary sizes $|D_D|$ gains overall impact

Outline

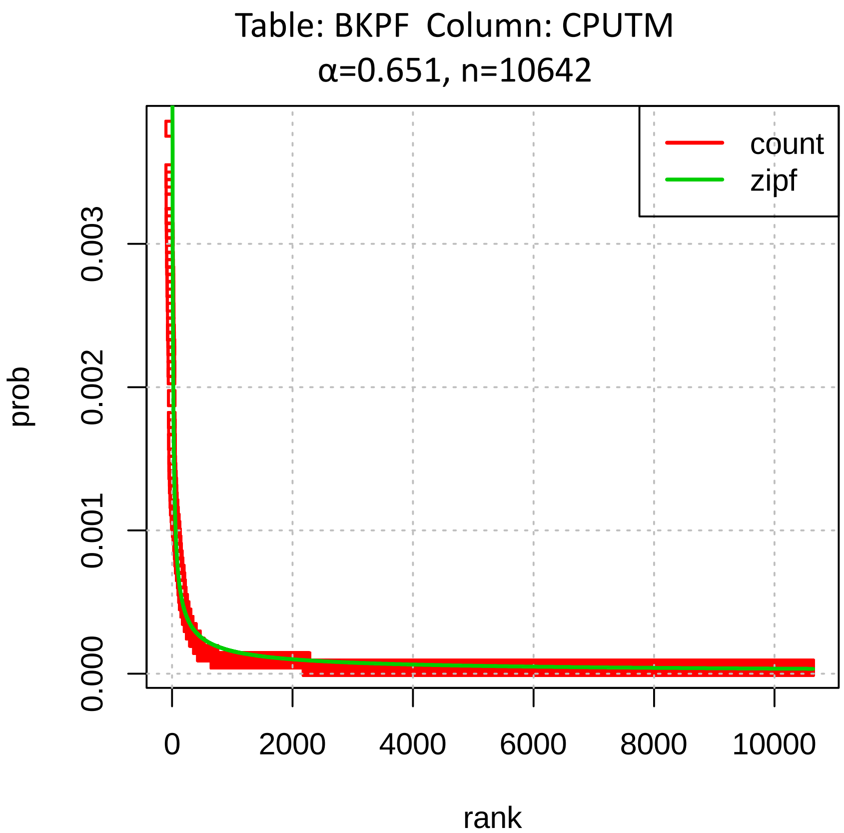
- Preliminaries
- **Value Distrib. in Enterprise Data**
- Partial Merge Algorithm
- Evaluation

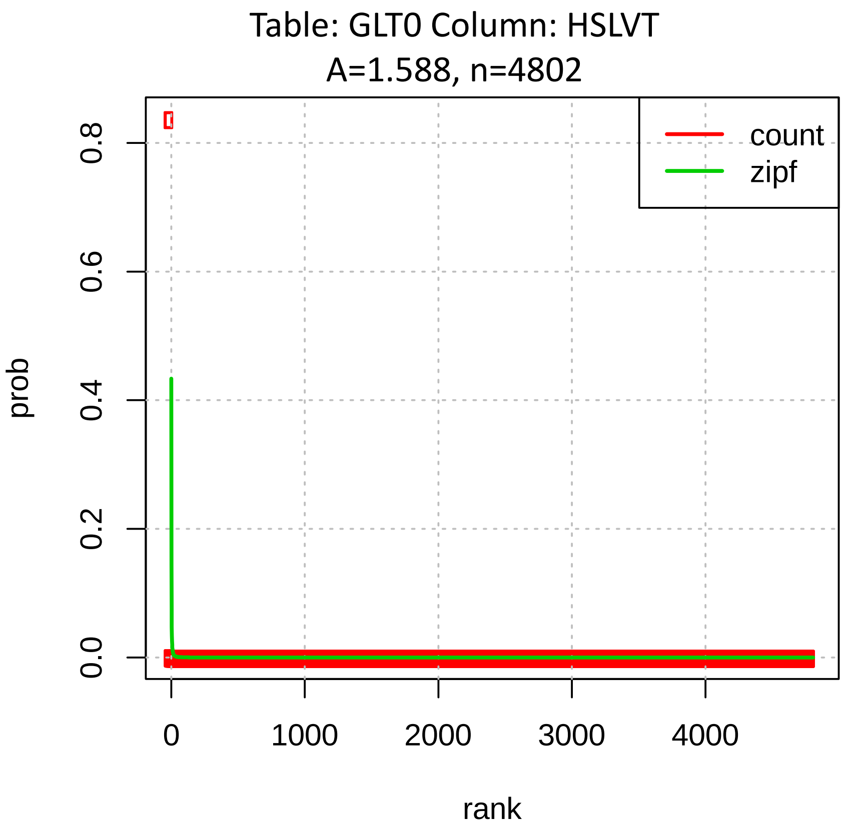
Distribution of Values in Enterprise Data

- We analyzed 1864 attributes (i.e. columns) from two SAP customer's Financial and Sales Distribution module of SAPs Business Suite
  – Count distinct values & create histograms
  – Match against PDFs for fixed domains
    - **ZIPF distribution** or
    - **Uniform distribution**
  – Ignore ID columns and single-value columns

PDF	Fraction	Parameter
zipf	21.03%	α_{min}=0.001, α_{Q2}=1.581, α_{max}=4.884
uniform	20.02%	k_{min}=2, k_{Q2}=5, k_{max}=216
single value or key	58,95%	-

Distribution of Values in Enterprise Data (Examples)





Zipf: most frequent values have a high probability to exist in D_M

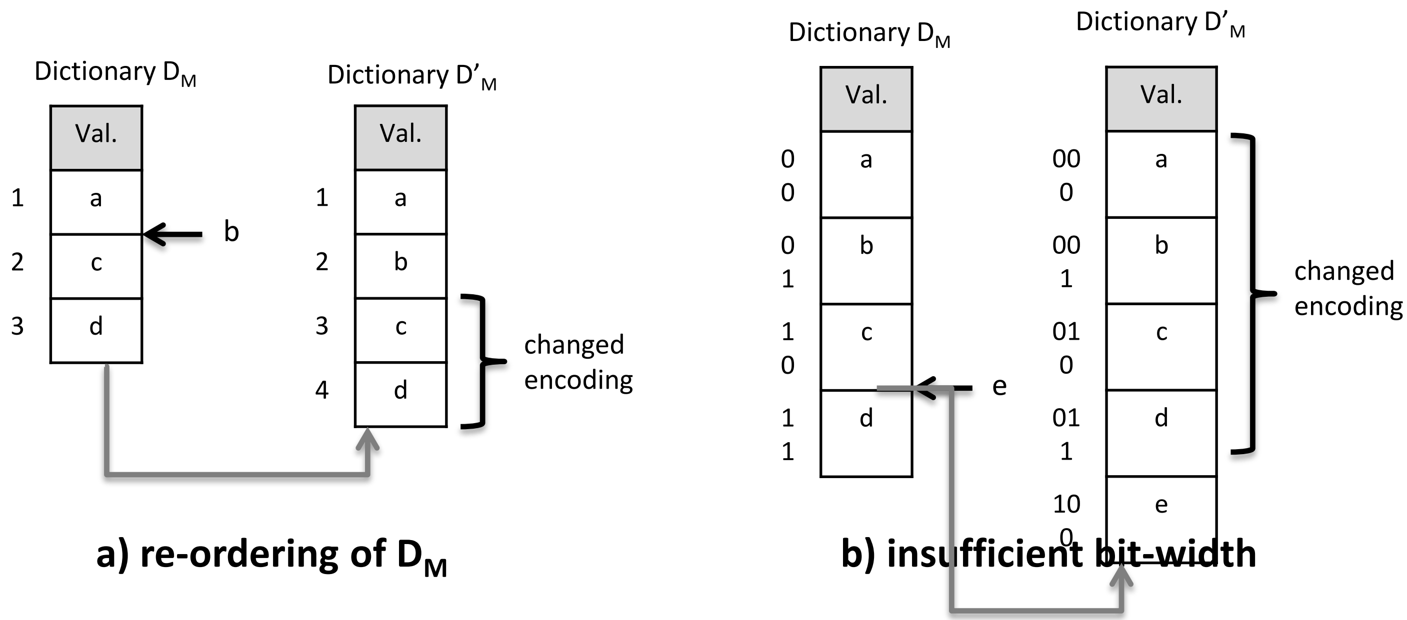
➔ **Data characteristics support partial merge strategy!**

Outline

- Preliminaries
- Value Distrib. in Enterprise Data
- **Partial Merge Algorithm**
- Evaluation

Full Merge – High complexity

- Merging all delta values:
  - Complexity in O($|C_M|+|C_D|$) if dictionary encoding changes
  - Change of dictionary encoding necessary if:
    - a) Main dictionary D_M is re-ordered (with high probability)
    - b) Bit-width of D_M not sufficient any more



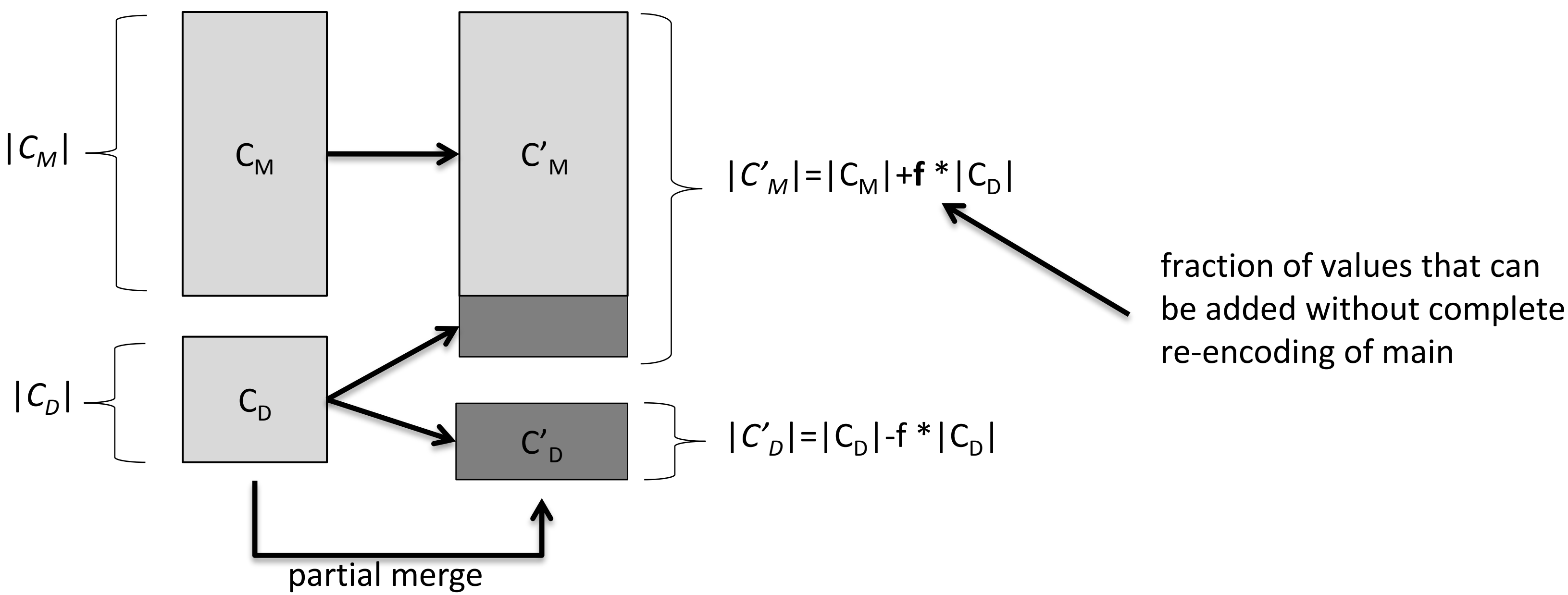
a) re-ordering of D_M

b) insufficient bit-width

- **Merge operation induces significant write overhead!**

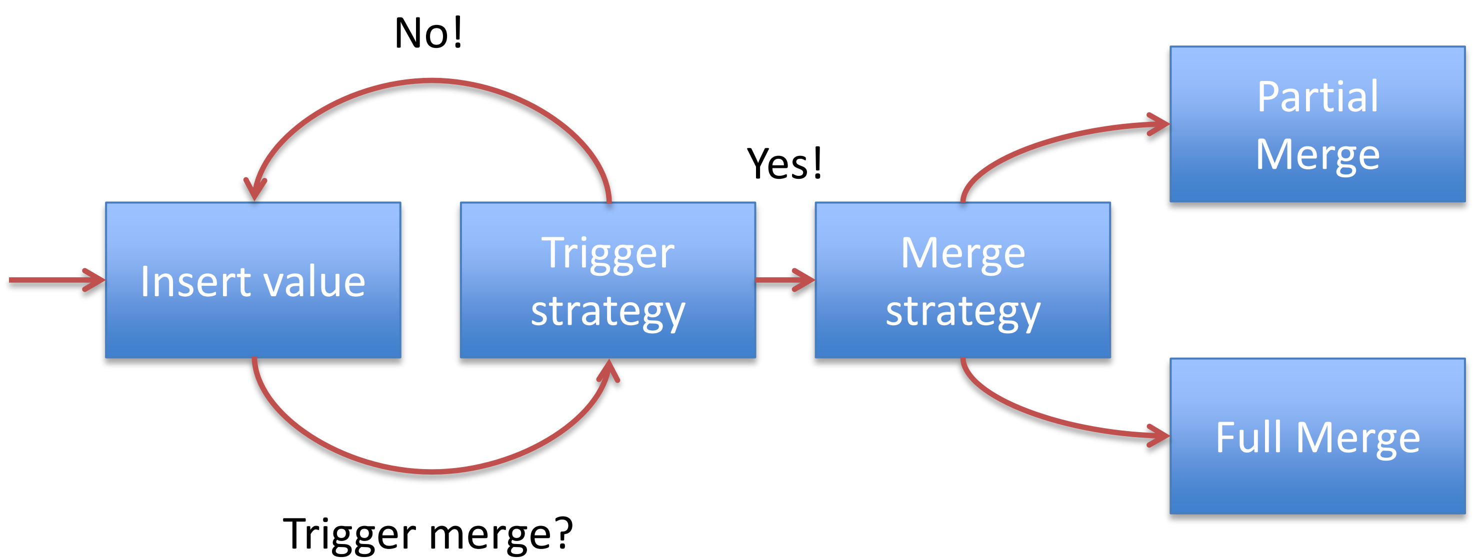
Partial Merge – Reduced Complexity

- ***Idea***: Merge only those values, that don't cause event (a), i.e., don't change the existing main encoding
  – Only add values from the delta that exist already in D_M or do not cause re-ordering of D_M



- Complexity now in $O(|C_D|)$ instead of $O(|C_M|+|C_D|)$ with $|C_D| << |C_M|$
- Expected size of fraction **f** depends on PDF of distinct values

The Partial Merge Algorithm



- Continuous process with **2 decisions**
  - Does the status of the delta store require a merge?
  - Is it feasible to conduct a partial merge?
- Strategies are aware of the read cost overhead and merge cost

Triggering Merge Operations possible with respect to cost

- **Fraction of optimum (*fro*) strategy:**
  - Trigger a merge after n-th insert, if current worst case select costs reach a defined distance f_{opt} from the theoretically opt. cost ($|C_D|=0$) (*selectCost^{opt}(n)*)

$$selectCost(n) > f_{opt} \cdot selectCost^{opt}(n)$$

  - Since *selectCost(n)* and *selectCost^{opt}(n)* are only defined by current size of the data structures, both can be easily computed at runtime

- Other strategies possible
  - Space consumption-aware
  - Workload aware
  - …

Cost-based decision on Partial Merge guarantees upper bound on overhead

- Values that remain in delta still cause performance overhead
  - When is this too much?

- **Cost aware strategy (*coa*)**: decide on merge strategy based on the possible improvement of **worst case range select** cost

- Partial merge is executed, if worst case range select cost after a partial merge ($selectCost_P$) are comparable to optimal cost after a full merge ($selectCost_F$), i.e. if :

$$selectCost_P \leq f_{to} \cdot selectCost_F$$

  with f_{to} being a factor representing the tolerated overhead

Outline

- Preliminaries
- Value Distrib. in Enterprise Data
- Partial Merge Algorithm
- **Evaluation**

Evaluation

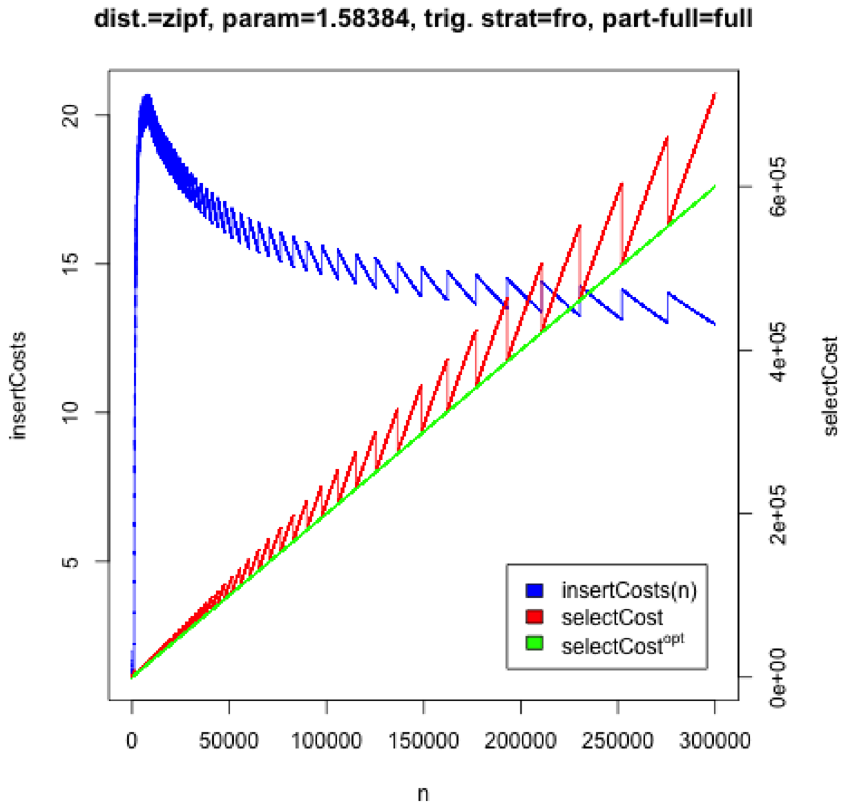
- Simulation of inserts (sampled from observed PDFs)
- Measure after every inserted value
  - ***selectCost(n)***: worst case range select
  - ***insertCost(n)***: amortized write cost (including potential merge)

$$insertCost(n) = \frac{\sum_{i=0}^{n} writes(v_i)}{n}$$

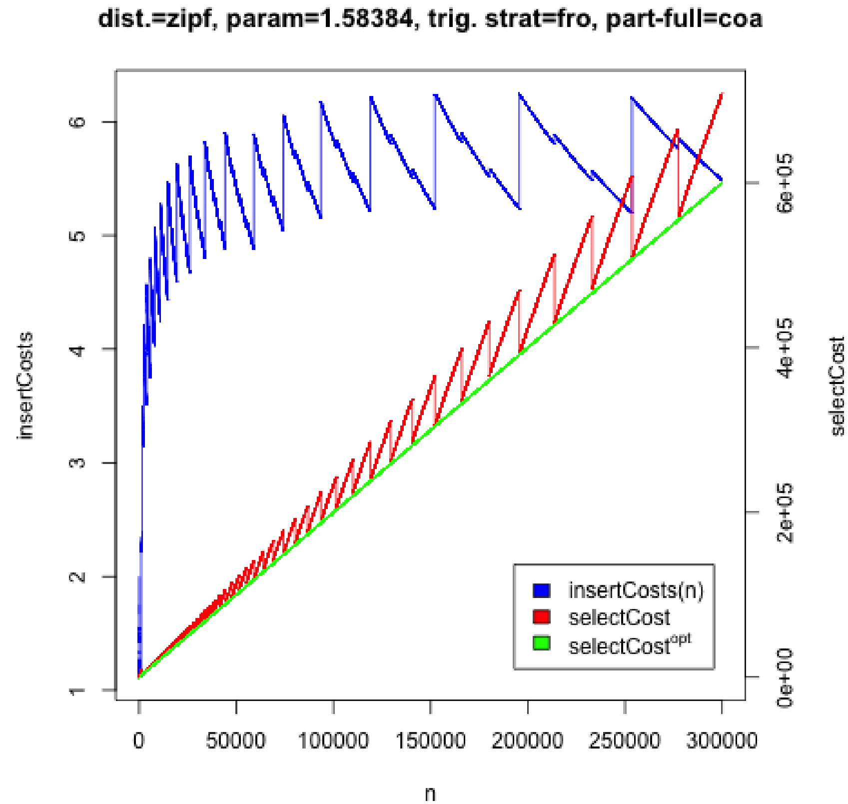
with writes(v_i) = #write operations required to insert v_i

Evaluation (1/3) – Zipf distributed distinct values

Full merge (always):



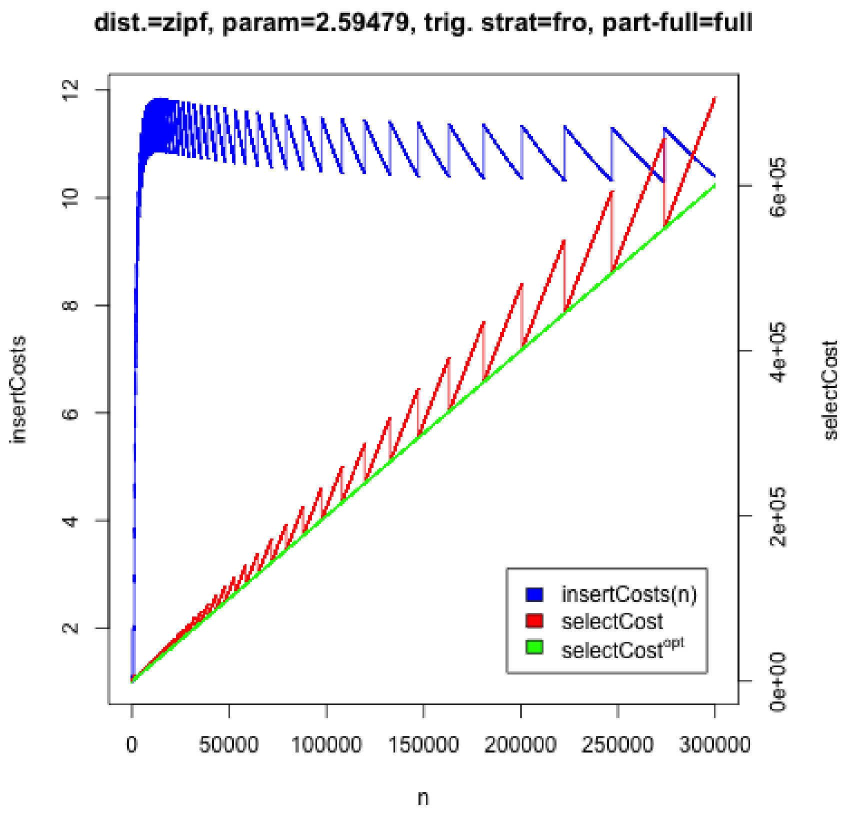
Partial merge (when possible):



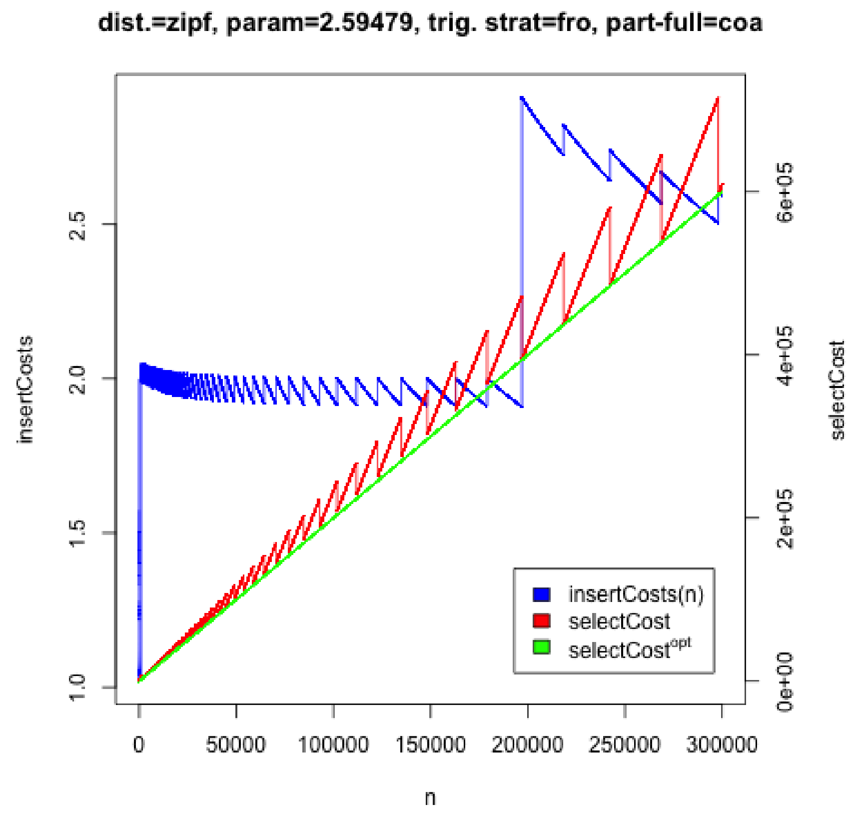
- Amortized insert performance improved > 100%

Evaluation (2/3) – Zipf distributed distinct values

Full merge (always):



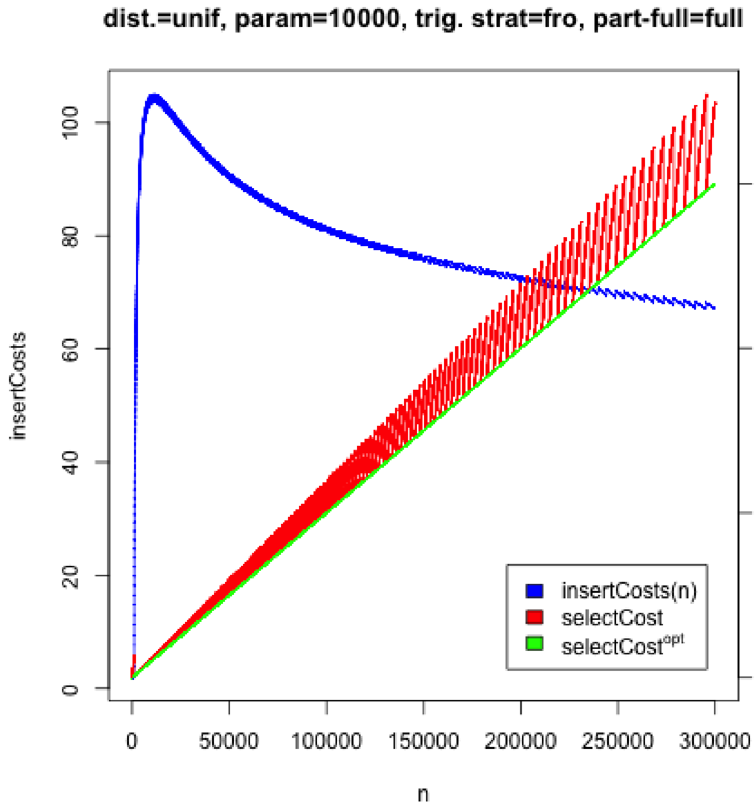
Partial merge (when possible):



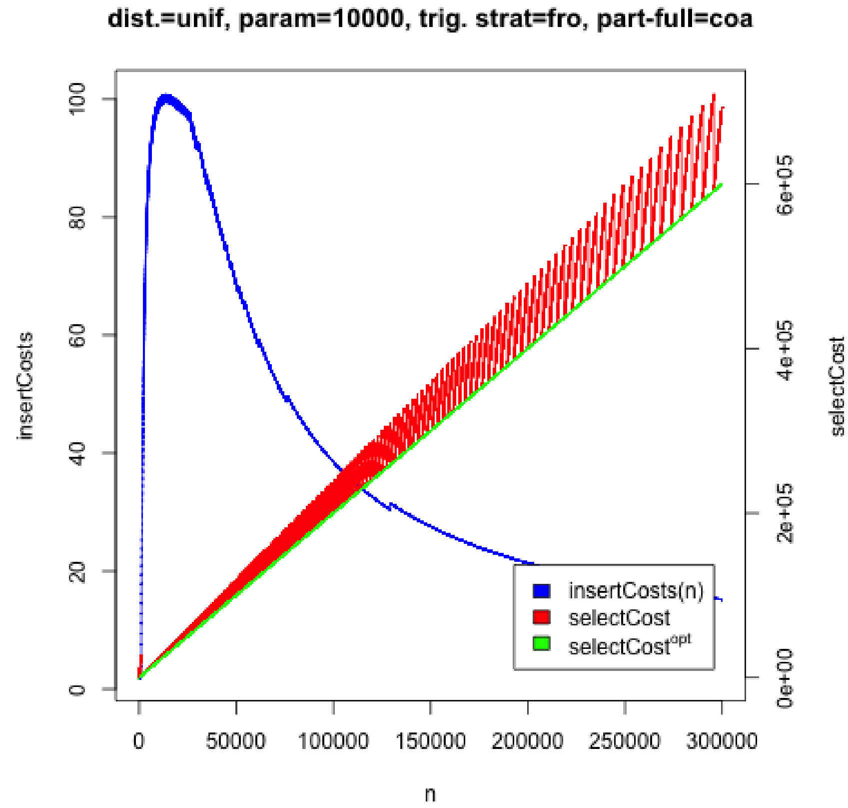
- Amortized insert performance improved > 400%

Evaluation (3/3) – Uniformly distributed distinct values

Full merge (always):



Partial merge (when possible):



- Amortized insert performance improved about 400%

Future Work

- Analyze potential for **tables** (vs. single columns)
- Find more sophisticated **strategies for trigger and merge**
- Take **workload** into account (and find cost minima)
- **Analyze** more customer data
- **Implement** for existing DB prototypes
- …

Thank You !