Processing a Trillion Cells per Mouse Click

NoSQL Matters 2013

Alex Hall, Google Zurich

Olaf Bachmann, Robert Buessow, Silviu Ganceanu, Marc Nunkesser

Outline of the Talk

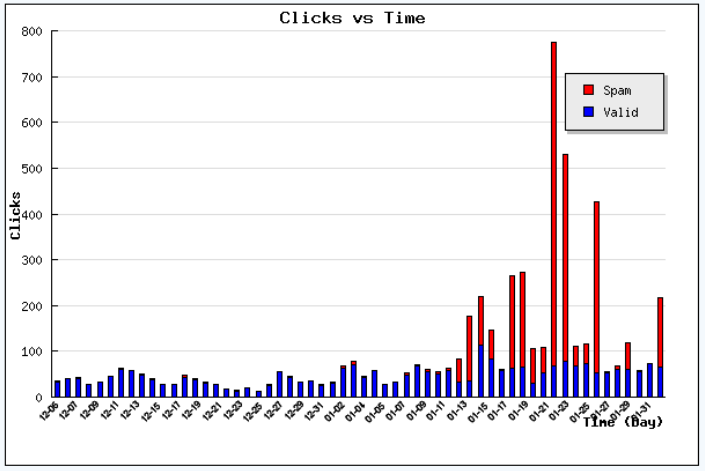
- Background: AdSpam team at Google
  Why we care about interactive data analysis
- PowerDrill UI: internal web-app to slice & dice data
- High-level properties of PD Serving
  scaling from millions to billions of records
- Main part: key ingredients, comparison to other backends, algorithmic engineering “tricks”

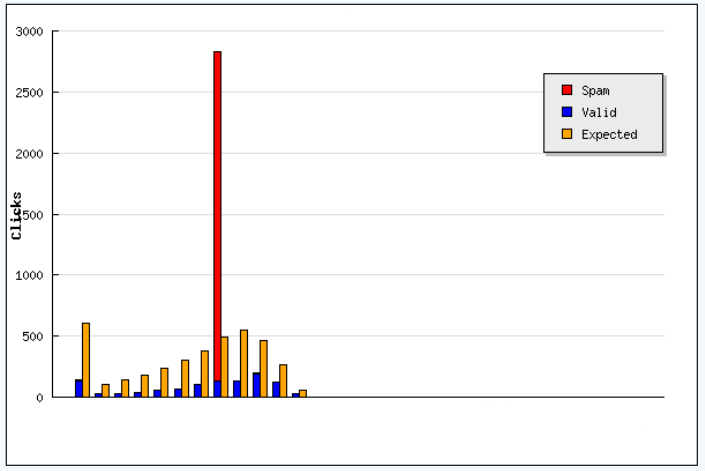
AdSpam: Interactive Data Analysis

AdSpam team provides online filters to catch "invalid clicks"

Typical analyses:

- Manually check set of suspicious clicks
- Slice and dice the data, look at various metrics





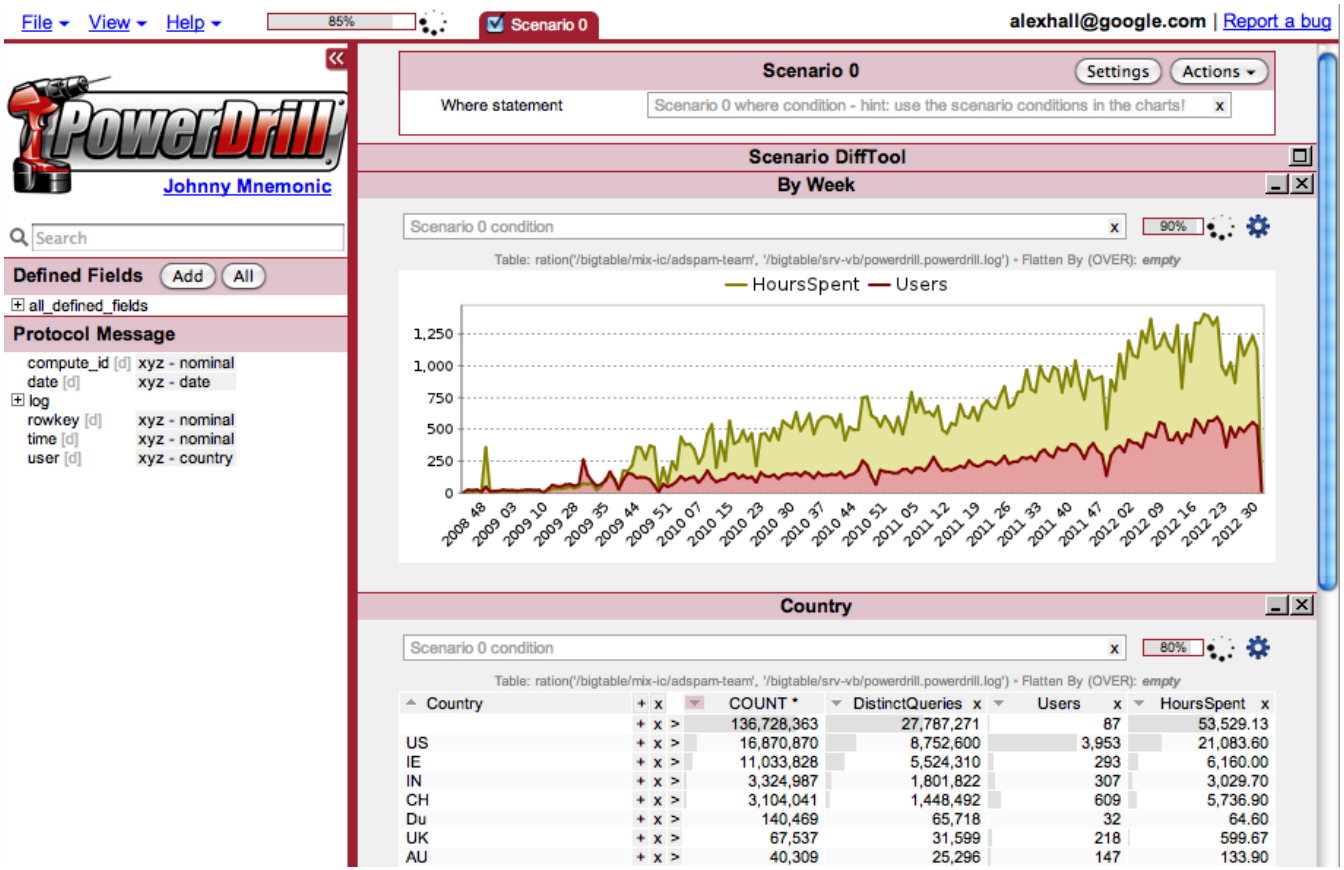
Goals:

- **Review: quickly decide whether clicks are invalid**
- **Filter development: research new filter ideas**

PowerDrill UI

Google internal web-app for easy slicing and dicing

- Shows charts e.g., clicks over time, top ten countries, …
- Interactive way of restricting the data set



PD logs (Google internal queries)

PowerDrill UI

Each chart -> SQL "GROUP BY" query
Restriction -> WHERE statement

On every interaction

- Send SQL queries to the backend
  Dremel, RecordIO, CSV, PD Serving, …
- Backend processes SQL on suspicious click data

Needs to be super fast on billions of records!

PD Serving's goal:
scale from 100s of millions to 10s of billions

wired.com

“Google Crunches One Trillion Pieces of Data With Single Click”

Appeared on www.wired.com/wiredenterprise/2012/08/google-trillion-pieces-of-data/

Contains nice summary of this VLDB article

> “Dremel is designed to analyze many different datasets,” says Tomer Shiran, […], “but this new system is optimized to run in memory, and that means you can achieve really, really low latency.” […]
> “If you have, say, four datasets that are central to your business,” Shiran says, “this is where you would store them.” The system uses various compression techniques, he says, to pack as much data as possible into memory.

Thoughts:

- Obviously, Google cares about more than four datasets
  Dremel: disk based, petabytes of data, millions of tables
- OTOH, AdSpam analyses: mostly with two huge (logs) datasets

Reality

- Heavily used within AdSpam since 2 years.

  Single user after a “hard day’s work”: up to 12k queries
- Used primarily on 2 major datasets
- Typically a single mouse click triggers 20 SQL queries
- **On average** these queries process data corresponding to 782 billion cells

  i.e., frequently > 1 trillion cells
- Return in 30-40 seconds (under 2 seconds per query)

Remainder of the Talk

- Data and queries for experiments
- Comparing existing backends (latency, mem)
- How to speed this up? Observations

  Indexed data vs. full scans, caches
- New approach, key ingredients
  - Get the best of both: indexes and full scans
  - Improve cache hits
- Optimizations / algorithmic engineering “tricks”

  Stepwise discussion of effects of optimizations

Data for Experiments

Our UI is used Google-wide, millions of SQL queries processed over the last half year

-> 5 mio records of own PD query logs for experiments ☺

country, city, office, user,
table_name [includes Dremel queries -> millions of tables],
latency, time_stamp

Experiments

- Ran on 2.6 GHz, 8GB RAM Linux machine
- 5 repetitions per query
- Disk-caches flushed between queries
- Measured avg latency and memory usage

Example SQL Queries

Q1: top 10 countries

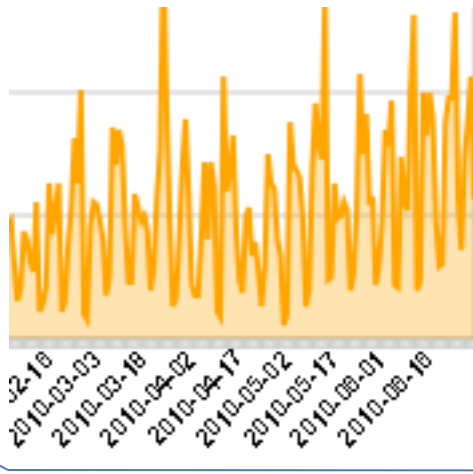
(users in 25 countries)

```
SELECT country, COUNT(*) as c
  FROM pd_logs
 GROUP BY country
 ORDER BY c DESC LIMIT 10;
```

US	+ x >	4,851
IN	+ x >	1,217
CH	+ x >	473
IE	+ x >	287
UK	+ x >	10
CN	+ x >	7
DE	+ x >	4
RU	+ x >	2
AU	+ x >	1
JP	+ x >	1

Q2: # of SQL queries, latency per day

```
SELECT date(time_stamp) as date,
       COUNT(*) as c, SUM(latency)
  FROM pd_logs
 GROUP BY date
 ORDER BY date ASC;
```



Q3: top 10 table-names from Ireland

```
SELECT table_name, COUNT(*) as c, SUM(latency)
  FROM pd_logs
 WHERE country = 'IE'
 GROUP BY table_name
 ORDER BY c DESC LIMIT 10;
```

Q4: top 10 table-names overall (w/o latency)

Comparing Existing Backends

- CSV files (comma separated values)

  Compute stats by iterating over a csv-file; scan whole file line-by-line

- RecordIO files

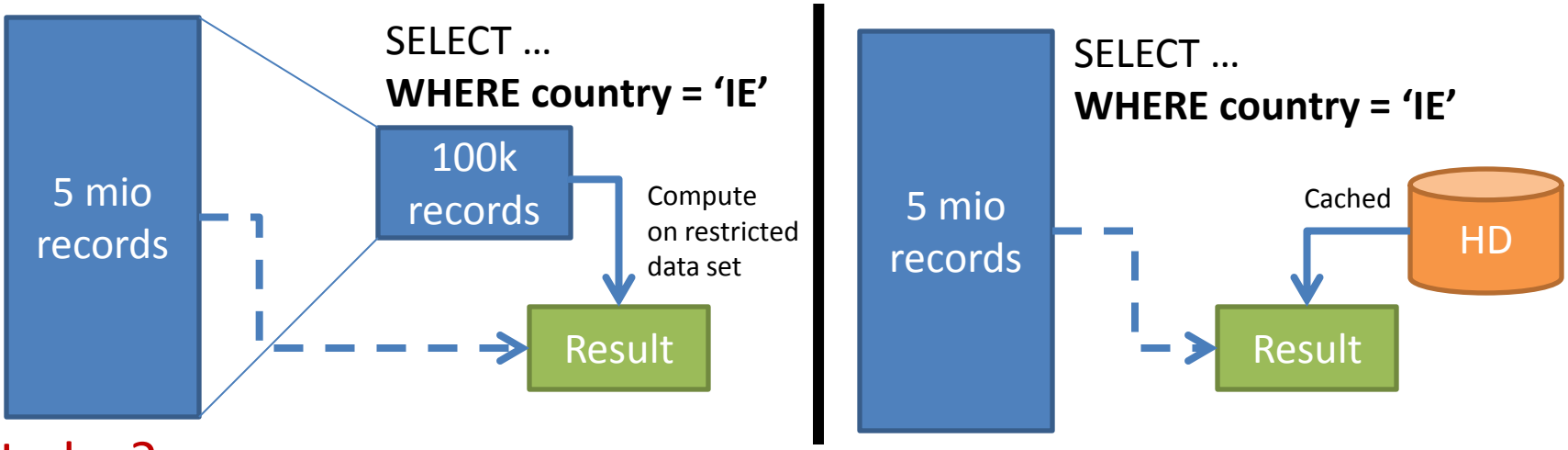
  Google binary "record" file-format; scan whole file record-by-record

- Dremel
  - High performance Google internal column store
  - Columnwise storage: full scan of data, but only necessary columns

	Latency in milliseconds				Memory in KB			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
CSV	55,099	75,207	52,924	71,778	573,339	573,339	573,339	573,339
RecordIO	27,134	50,587	28,497	39,235	551,074	551,074	551,074	551,074
Dremel	7,874	18,191	8,907	48,628	27,943	60,369	118,734	90,792

Obervations

- Columnwise full scans are very fast! Cache locality, good to opt…
- Would be nice to skip data though …

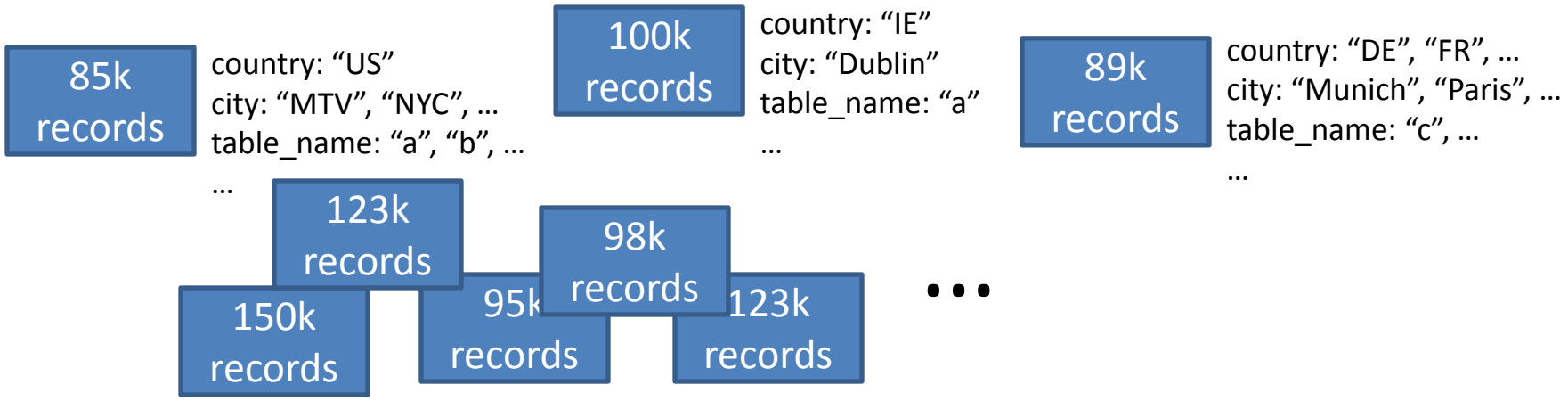


- Index?
  - Fixed set of fields (only for certain WHERE restrictions)
  - Expensive to evaluate compared to full scan
    DBs like SQL Server do full scans if more that 10% of data touched
- Caches?
  - Insufficient because too much variance on the queries

Best of Both: Index vs. Full Scans

- Partition the data during import (composite range partitioning)
- Add "index" per chunk: per field a list of occurring values

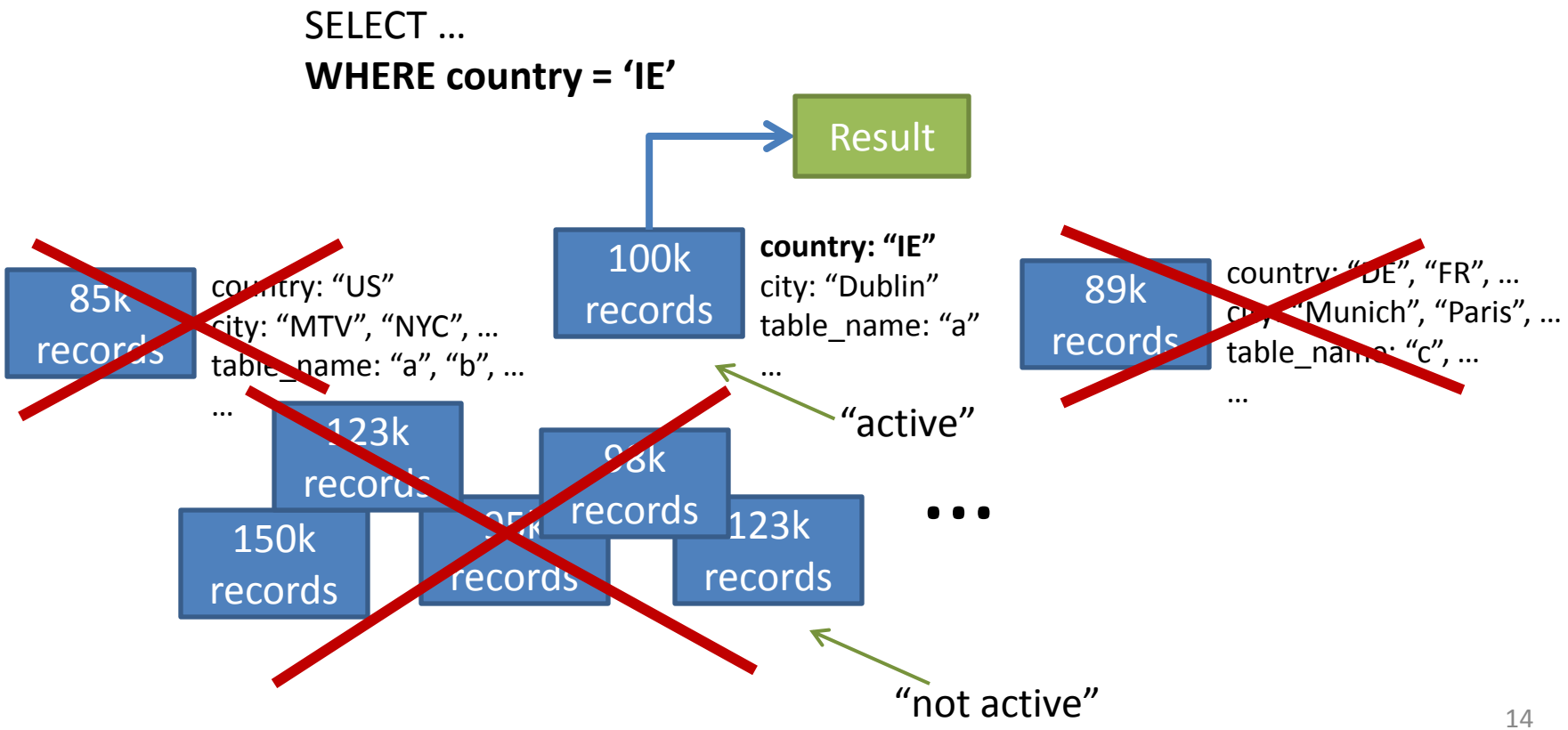
-> WHERE restricts chunks, fast columnwise scan per chunk



Best of Both: Index vs. Full Scans

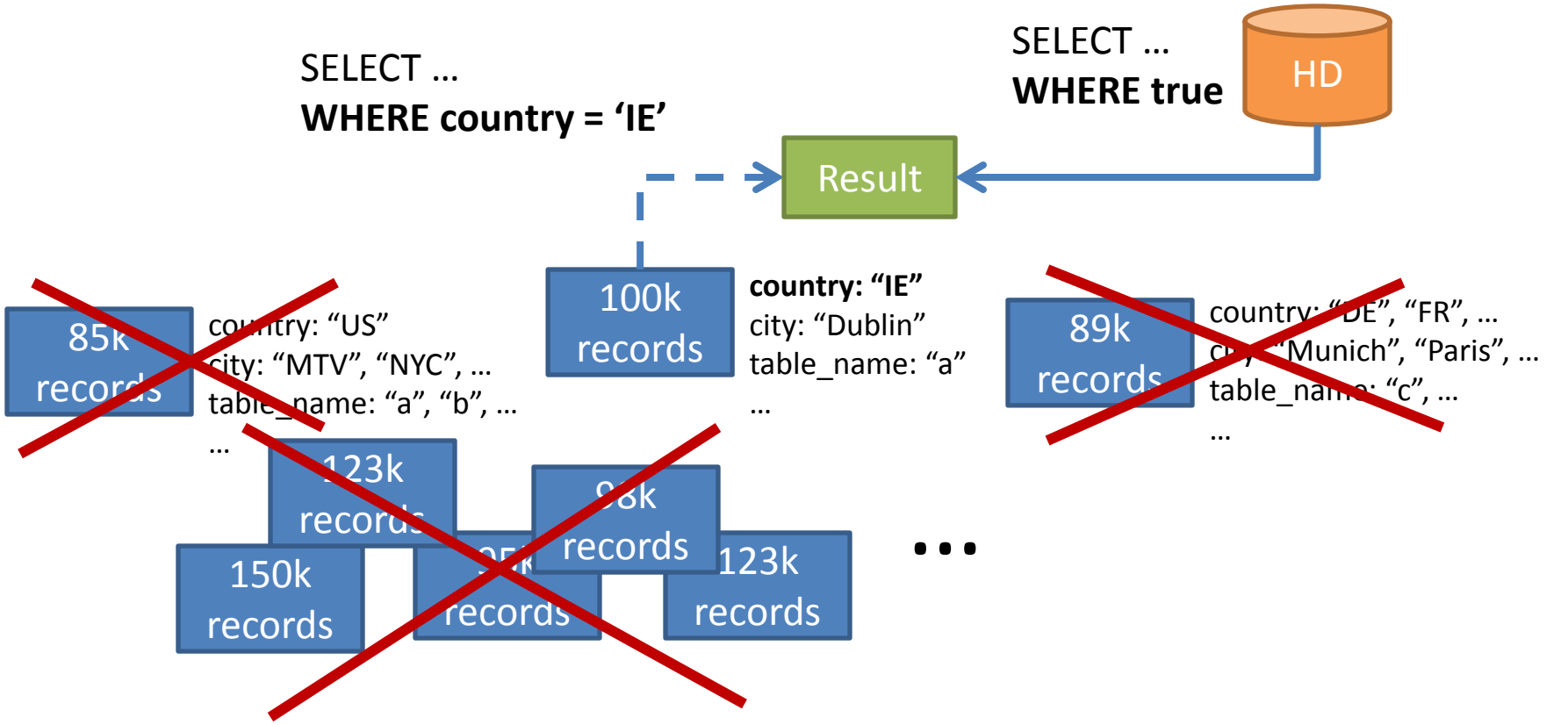
- Partition the data during import (composite range partitioning)
- Add "index" per chunk: per field a list of occurring values

-> WHERE restricts chunks, fast columnwise scan per chunk



Improve Cache Hits

- Cache result per chunk
- "Normalize" WHERE statement per chunk, e.g.,
  Chunk contains only **country = 'IE'** -> remove WHERE



Outline of Experiments

Stepwise check individual improvements

Measure latency & memory usage

- Basic system
  - **No** partitioning, **no** index, **no** caches
  - Data structures / format optimized for GROUP BY and IN restrictions
- Partition: Split data into chunks, add index
- Further optimizations / “tricks” to reduce memory usage
  Goal: serve as much from memory as possible

Basic System

- **No** partitioning, **no** index, **no** caches
- Columnwise storage, per field store:
  - Dictionary: occurring values <-> int “ids”
  - Represent the actual data as list of such ids

	Latency in milliseconds				Memory in KB			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
Dremel	7,874	18,191	8,907	48,628	27,943	60,369	118,734	90,792
Basic	20	214	179	686	20,001	41,453	132,682	91,232

Q1 Top countries	Q2 Count & latency / day	Q3 Top tables	Q4 Top tables
-> 5 mio times counts[countryId]++	pre-computed date(..)	WHERE restriction many values / ids	no WHERE many values / ids

Partition the Data and Add Index

- Split data into chunks (simple composite range partitioning)
- Add index (i.e., list of occurring int ids) to each chunk

	Latency in milliseconds				Memory in KB			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
Dremel	7,874	18,191	8,907	48,628	27,943	60,369	118,734	90,792
Basic	20	214	179	686	20,001	41,453	132,682	91,232
Index	19	226	42	618	20,073	47,986	69,808	91,317

Q3 Top tables
WHERE restriction
many values / ids

Reduce Memory Footprint

Goal: billions of rows in memory

Optimized Storage of **int ids**

- Until now: store ids as 4 byte ints.
- Low hanging fruit: select optimized storage **per chunk**
  - 1 id (2, 2, 2, … for “US”, “US”, “US”, …) -> O(1) storage
  - 2 ids (0, 2, 2, … for “DE”, “US”, “US”, …) -> Bitset
  - < 2^8, < 2^16, < 2^32 ids resp. -> 1, 2, 4 bytes/id

Memory usage of int ids in KB

	Q1	Q2	Q3	Q4
Basic	20,001	40,726	64,934	24,209
Chunks	20,072	47,259	51,426	24,293
Index	20,072	47,259	2,060	24,293
OptCols	83	22,259	1,266	14,292

Only 25 countries. Many chunks with just one / two countries

WHERE restricts # chunks read

Top tables. Many ids

Optimized **int ids** – Latency & Memory

- Latency basically unchanged
- A lot less memory for fields with few ids
- Gains not as large for fields with many ids (large dictionaries!)

	Latency in milliseconds				Memory in KB			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
Dremel	7,874	18,191	8,907	48,628	27,943	60,369	118,734	90,792
Basic	20	214	179	686	20,001	41,453	132,682	91,232
Chunks	31	231	99	636	20,073	47,986	119,174	91,317
Index	19	226	42	618	20,073	47,986	69,808	91,317
OptCols	20	297	43	659	83	22,986	69,014	81,315

Only 25 countries.
Many chunks with just one / two countries

Top tables.
Many values / ids

Optimized **Dictionaries**

- Dictionaries can be large, e.g., verbatim all table-names
- Sorted, often long common prefixes!
- Optimized Trie (heavy on bit manipulations on single byte[])
- Each node corresponds to a “nibble” (4 bits)
- Lookup in both directions (2 -> “US” and “IE” -> 1)

Memory Dictionaries in KB

	Q1	Q2	Q3	Q4
Basic	0	727	67,748	67,023
Chunks	0	727	67,748	67,023
Index	0	727	67,748	67,023
OptCols	0	727	67,748	67,023
OptMaps	0	725	4,090	3,365

Only 25 countries

Many ids / string values with shared prefixes. Common case in practice

Optimized **Dictionaries** - Overall

- Latency more or less the same
- Huge improvement for fields with many ids
- Optimized, in-memory data structures:
  5x smaller than Dremel's compressed version.

	Latency in milliseconds				Memory in KB			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
Dremel	7,874	18,191	8,907	48,628	27,943	60,369	118,734	90,792
Basic	20	214	179	686	20,001	41,453	132,682	91,232
Chunks	31	231	99	636	20,073	47,986	119,174	91,317
Index	19	226	42	618	20,073	47,986	69,808	91,317
OptCols	20	297	43	659	83	22,986	69,014	81,315
OptMaps	17	373	42	653	83	22,984	5,356	17,658

Compressed – **Snappy**

- To squeeze more into the RAM, how about compressing?
- Google's own fast Snappy algorithm
- Focus on Q4 -- the many int ids / values case

	Memory in KB				Compressed Memory in KB			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
Basic	20,001	41,453	132,682	91,232	3,018	17,347	31,389	17,699
Chunks	20,073	47,986	119,174	91,317	279	16,340	23,965	12,194
OptCols	83	22,986	69,014	81,315	42	16,321	2,688	12,193
OptMaps	83	22,984	5,356	17,658	42	16,321	2,889	12,395

- Partitioning makes a difference (chunks have "similar" values)
- Almost as if a wall is hit at 12MB with "byte-level techniques"

Note: Huffman coding techniques go further ("bit-level")

Snappy & Reordering

Snappy

- Additional 29% – 49% saving in compression
- About 2x worse latency!

-> Hybrid: two in-memory layers (compr. vs. not compr.)
treat uncompressed with cache heuristic (e.g., LRU)

Reordering of rows to help compression

- Additional saving of up to 64% for int ids after compression

Compression Savings per Step

	Savings (per step)
Basic (compared to Dremel)	-11% – 34%
Chunks (partitioning)	-16% – 0%
Optimized storage of int ids	11% – 99.5%
Optimized dictionaries (trie)	0% – 78%
Snappy	29% – 49%
Reorder	16% – 55%

Summary

- Latency

  Reduced from 7-48 **seconds** to 7-260 **milliseconds**

- Memory

  From 27, 60, 90 MB down to 35KB, 12MB, 5.6MB

- **In production**, on average
  - 30-40 seconds for about 20 queries
  - 92.41% of records skipped
    5.02% served from cached results
    2.66% scanned
  - 70% of queries fetch no data from disk,
    96.5% less than 1GB (overall)