Big Data at Google and How to Process a Trillion Cells per Mouse Click

LSDMA Karlsruhe 24.9.2013

Alex Hall

Overview

- Big Data Analysis “Mainstream at Google”
    - MapReduce (MR)
    - Sawzall
    - Dremel

- AdSpam Team
  Why do we care about interactive data analysis

- PowerDrill UI: internal web-app to slice & dice data

- PowerDrill Serving: in-memory column-store
  scaling from millions to billions of rows

MapReduce: Basics

2003: Jeff Dean, Sanjay Ghemawat

Map: input ==> (key, value)
Shuffle
Reduce: (key, [values]) ==> output

Number of queries:
Map: WebSearch ==> (query, 1)
Reduce: (query, [1,...,1])==> query, len([1,..,1])

MapReduce: Shine and Whine

Shine:

- Write a simple program, run on 10k machines
- Process data at 50GB/s
- No need to have experience with parallel systems
- Flexibility/low-level control via cmd-line opts

Whine:

- Need MR-foo to debug/optimize
- A lot of boilerplate/repetitions (think: sums)
- Not interactive -- slow “discovery cycles”
- Engineers-only tool (C++, etc)

MR Memegen

Sawzall: Basics

2005: Rob Pike et. all

szl: scripting language for MR

Map:

given an input record --
szl script describes how to emit to "aggregators"

Reduce:

10+ system-provided aggregators

sum, maximum, quantile, sample, unique, top

Sawzall: Example

```
proto "WebSearch.proto"
search: WebSearch = input;
num_per_ip: table sum[ip: string] of count: int;
if (search.query == "flowers") {
  emit num_per_ip[format_ip(search.ip_v4)] <- 1;
}
```

```
$ saw --program example.szl \
      --input_files /gfs/cluster1/websearch/2012/05/28/websearch/*.recordio
      --destination /gfs/cluster2/$USER/nperip@100
```

```
$ dump --source /gfs/cluster2/$USER/nperip@100
  --------------------
  | ip         | count|
  --------------------
  | "1.2.3.4" |  23  |
  | "3.4.5.6" | 736  |
  | "6.7.8.9" |  42  |
  |        ...       |
  --------------------
```

Sawzall: Shine and Whine

Shine:

- Sawzall scripting instead of C++ & gcc
- Powerful & extensive library and aggregators
- Users: engineers, product managers and analysts

Whine:

- No built-in/awkward chaining
- Not interactive (same as MR)

Sawzall Memegen

Dremel: Basics

2006/2010: Sergey Melnik, Andrey Gubarev

What

- Represents records (proto-buffers) as column-store
- Petabytes of data, millions of tables
- SQL as query-language

How

- Streams from disk
- Thousands of light-weight servers (leaves)
- Mixer-tree which aggregates intermediate results

Interactive query results for 100s of millions of rows

Dremel: Example

```
$ dremel
> SELECT format_ip(ip_v4) as ip,
         count(*) as count
    FROM "/gfs/cluster1/websearch/2012/05/28/websearch/*.columnio"
   WHERE query = "flowers"
   GROUP BY ip;
  --------------------
  | ip        | count|
  --------------------
  | "1.2.3.4" |  23  |
  | "3.4.5.6" | 736  |
  | "6.7.8.9" |  42  |
  |        ...       |
  --------------------
```

Dremel: Shine and Whine

Shine:

- Interactive data analysis over millions records
- SQL & CLI instead of szl & saw
- users: all Googlers (including sales folks)

Whine:

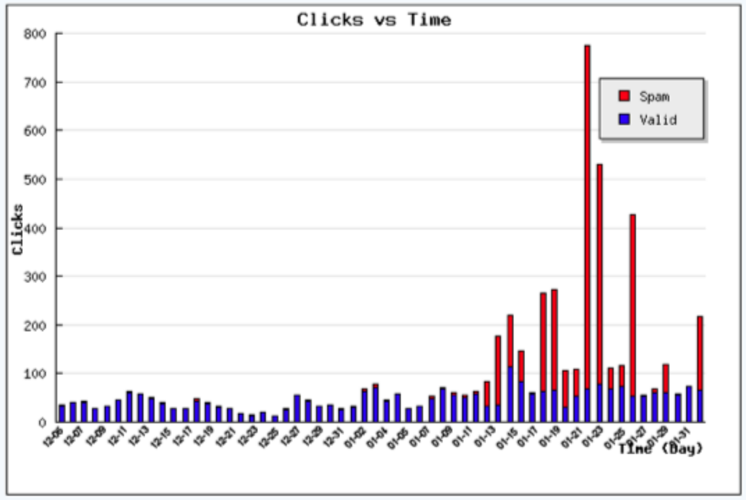
- No graphical analysis tool
- Scale! What if we want to go over billions of records

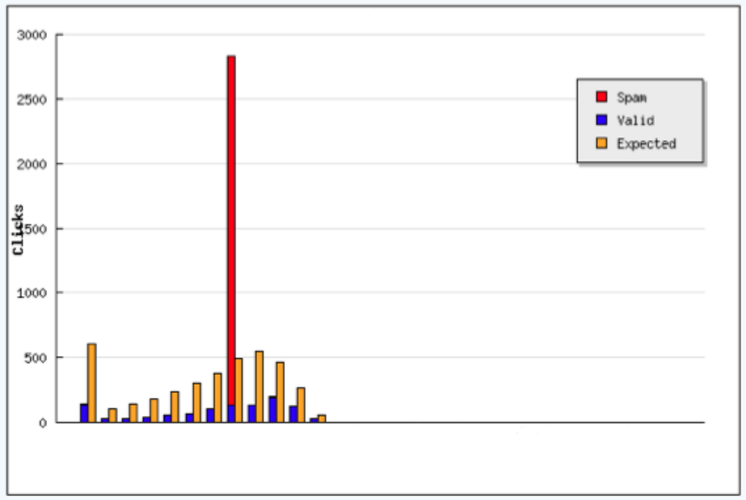
AdSpam: Interactive Data Analysis

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





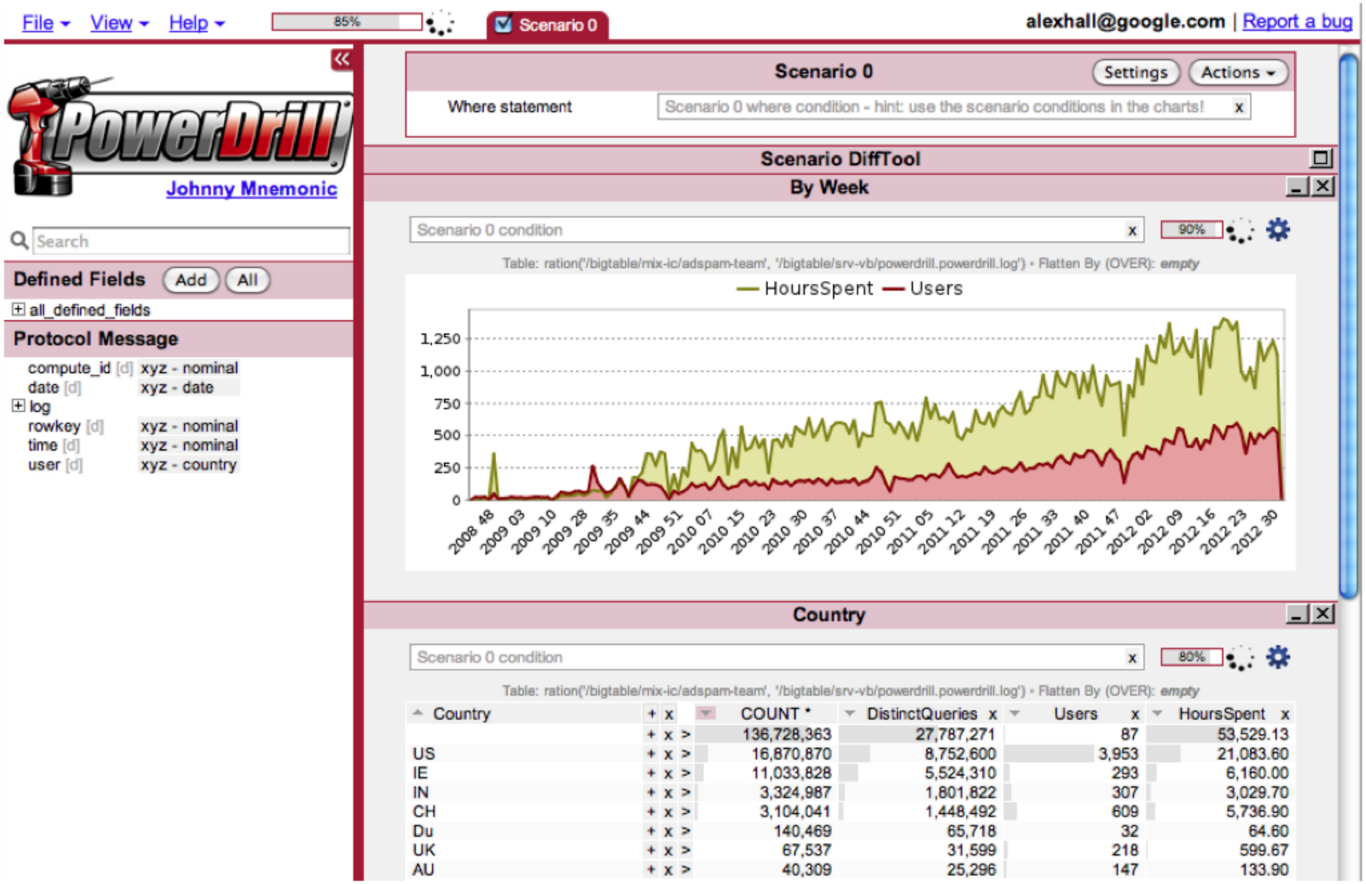
Goals:

- **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



Scenario 0

Where statement: Scenario 0 where condition - hint: use the scenario conditions in the charts!

⊞ all_defined_fields

Protocol Message

compute_id [d]	xyz - nominal
date [d]	xyz - date
⊞ log	
rowkey [d]	xyz - nominal
time [d]	xyz - nominal
user [d]	xyz - country

Scenario 0 condition

Table: ration('/bigtable/mix-ic/adspam-team', '/bigtable/srv-vb/powerdrill.powerdrill.log') • Flatten By (OVER): *empty*

Scenario 0 condition

Table: ration('/bigtable/mix-ic/adspam-team', '/bigtable/srv-vb/powerdrill.powerdrill.log') • Flatten By (OVER): *empty*

Country		COUNT *	DistinctQueries	Users	HoursSpent
	+ x >	136,728,363	27,787,271	87	53,529.13
US	+ x >	16,870,870	8,752,600	3,953	21,083.60
IE	+ x >	11,033,828	5,524,310	293	6,160.00
IN	+ x >	3,324,987	1,801,822	307	3,029.70
CH	+ x >	3,104,041	1,448,492	609	5,736.90
Du	+ x >	140,469	65,718	32	64.60
UK	+ x >	67,537	31,599	218	599.67
AU	+ x >	40,309	25,296	147	133.90

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, PowerDrill Serving, CSV, RecordIO, ...
- Backend processes SQL on suspicious click data

Needs to be super fast on billions of records!

PowerDrill Serving

Dremel

- Column-store, streams from disk
- Petabytes of data, millions of tables
- Thousands of light-weight servers
- Fast for 100s of millions of rows

PowerDrill Serving

- In-memory column-store
  “as much as possible” in-memory
- Few selected data-sets
- ~1500 servers, 6 TB ram
- **Scale to 10s of billions of rows**

VLDB 2012

PowerDrill Serving -- Usage in AdSpam

- Heavily used within AdSpam since 3 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

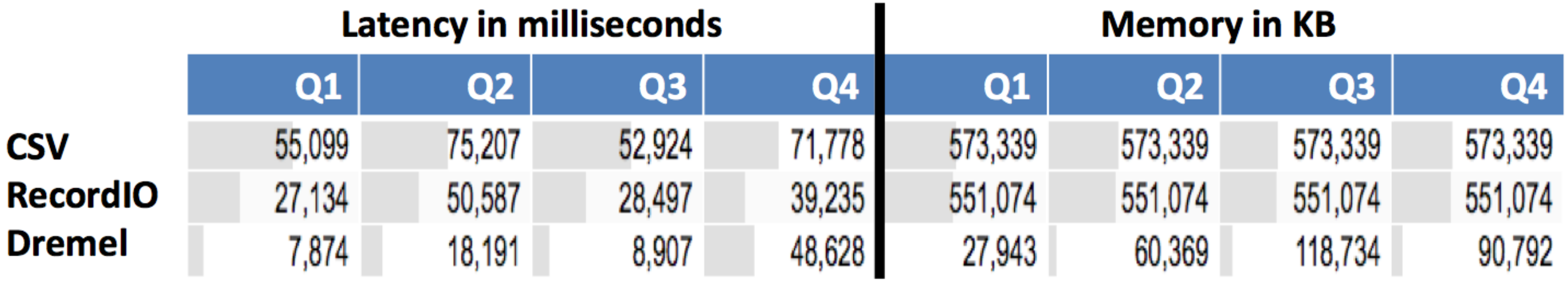
- Comparing existing backends/formats (latency, mem)
- Basic data-structures
- Key ideas: skipping data & cacheing
- Optimizations/algorithmic engineering “tricks”
  Stepwise discussion of effects of optimizations
- Performance in practice

Comparing Existing Backends/Formats

- **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**
  Columnwise storage: full scan of data, but only necessary columns

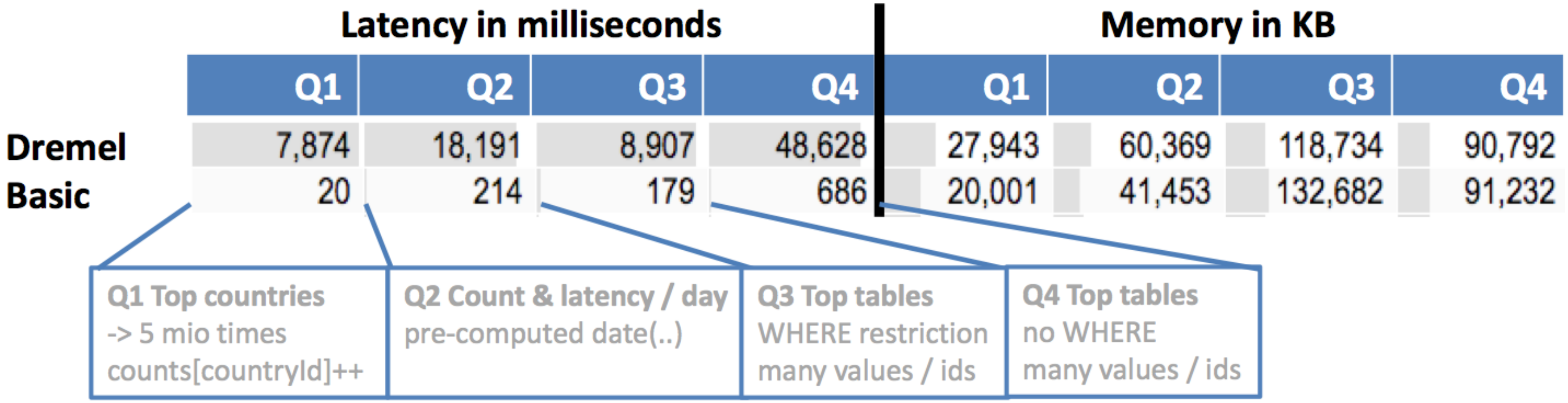


	Latency in milliseconds				Memory in KB			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
CSV	55,099	75,207	52,924	71,778	573,339	573,339	573,339	573,339
RecordIO	27,134	50,587	28,497	39,235	551,074	551,074	551,074	551,074
Dremel	7,874	18,191	8,907	48,628	27,943	60,369	118,734	90,792

PD Serving -- Basic Data-Structures

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
-> 5 mio times
counts[countryId]++

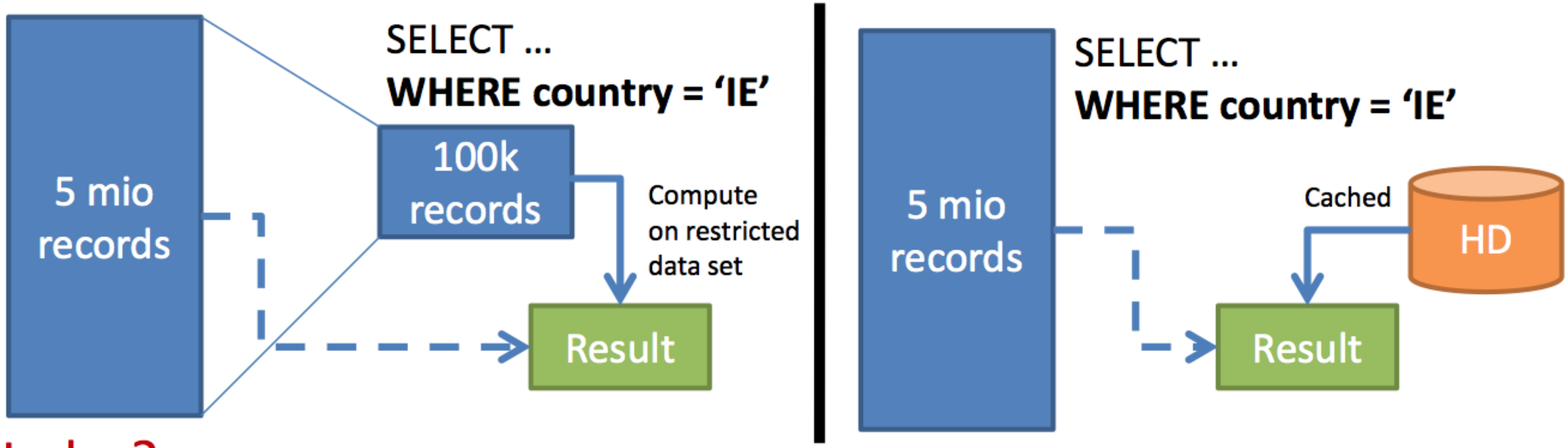
Q2 Count & latency / day
pre-computed date(..)

Q3 Top tables
WHERE restriction
many values / ids

Q4 Top tables
no WHERE
many values / ids

Observations

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

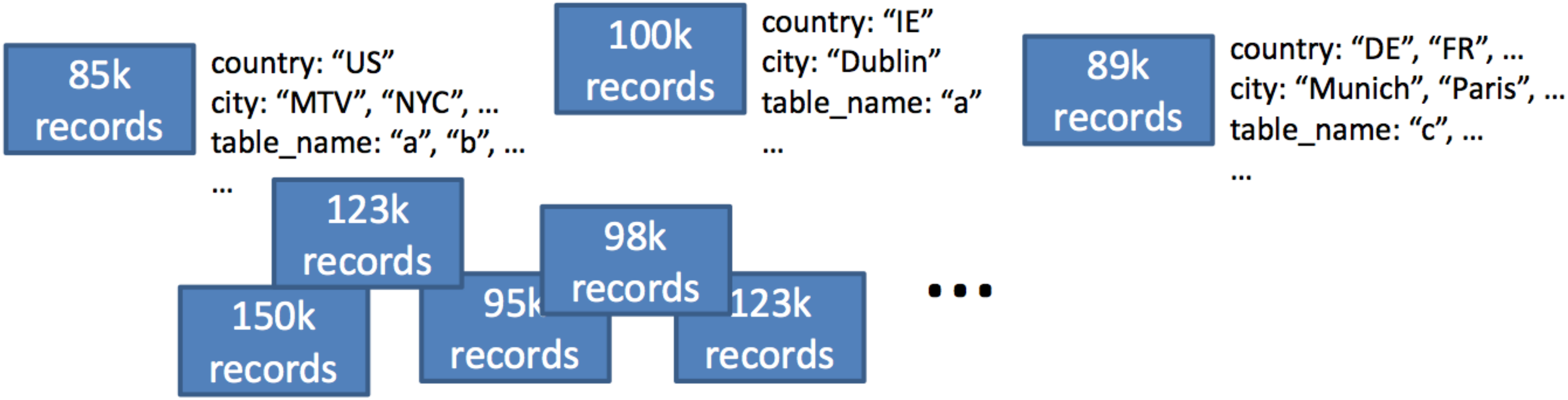


- Index?
  - Fixed set of fields (only for certain WHERE restrictions)
  - Expensive to evaluate compared to full scan
    DBs like SQL Server do full scans if more that 10% of data touched
- Caches?
  - Insufficient because too much variance on the queries

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



85k records

country: “US”
city: “MTV”, “NYC”, …
table_name: “a”, “b”, …
…

100k records

country: “IE”
city: “Dublin”
table_name: “a”
…

89k records

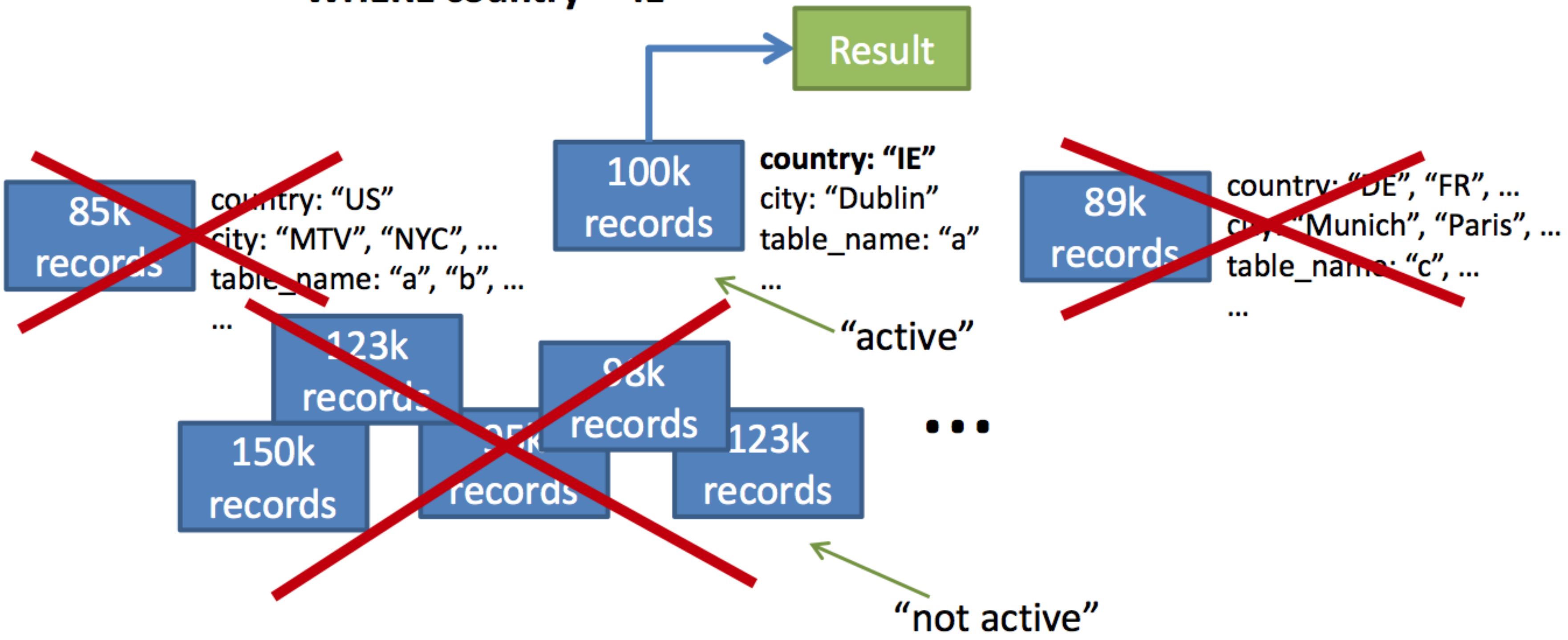
country: “DE”, “FR”, …
city: “Munich”, “Paris”, …
table_name: “c”, …
…

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

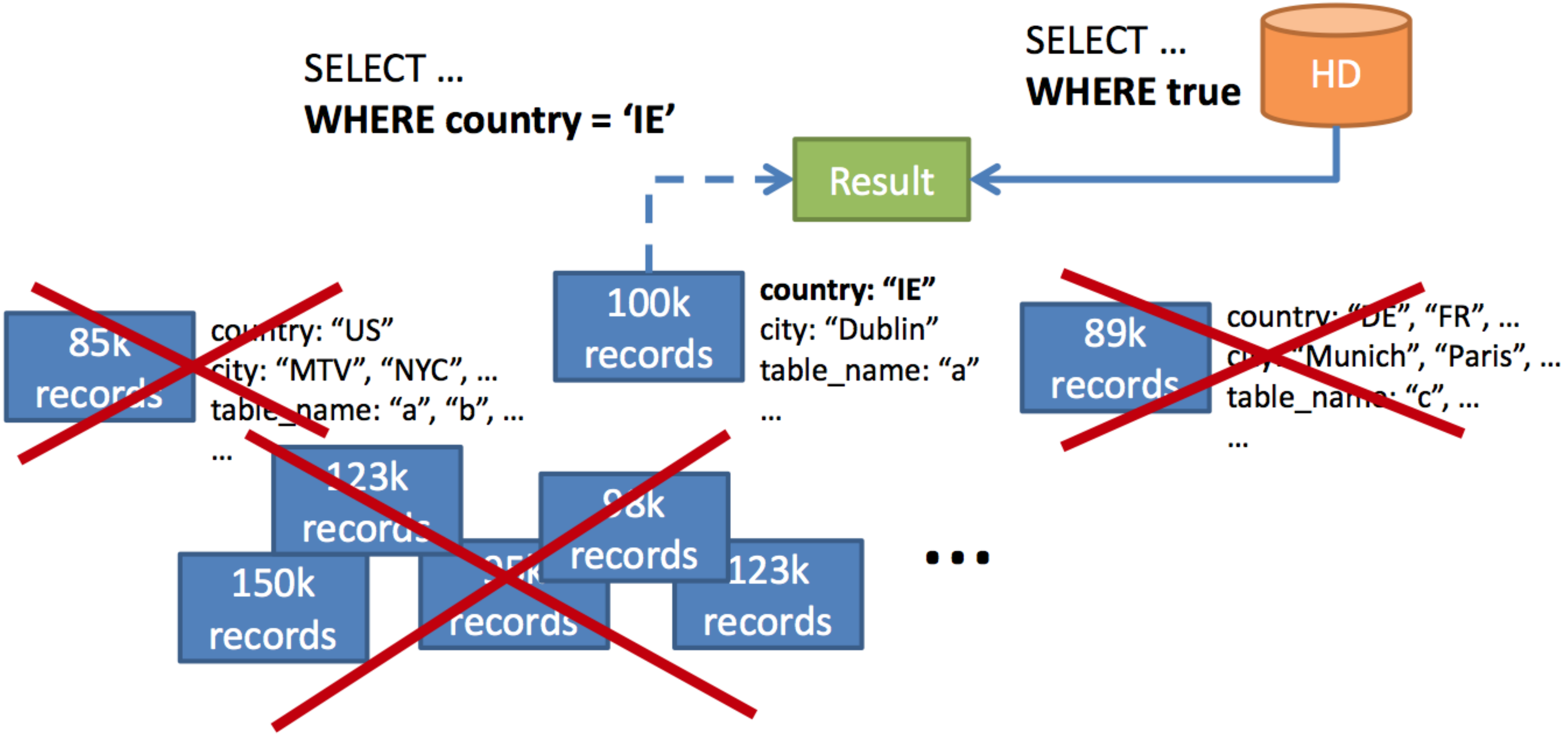
=> WHERE restricts chunks, fast columnwise scan per chunk

```
SELECT …
WHERE country = ‘IE’
```



Improve Cache Hits

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



SELECT ...
WHERE country = 'IE'

SELECT ...
WHERE true

HD

Result

100k records

country: "IE"
city: "Dublin"
table_name: "a"
...

85k records

country: "US"
city: "MTV", "NYC", ...
table_name: "a", "b", ...
...

89k records

country: "DE", "FR", ...
city: "Munich", "Paris", ...
table_name: "c", ...
...

123k records

98k records

150k records

records

123k records

...

Reduce Memory Footprint

Goal: Billions of rows in memory

Compression Savings per Step

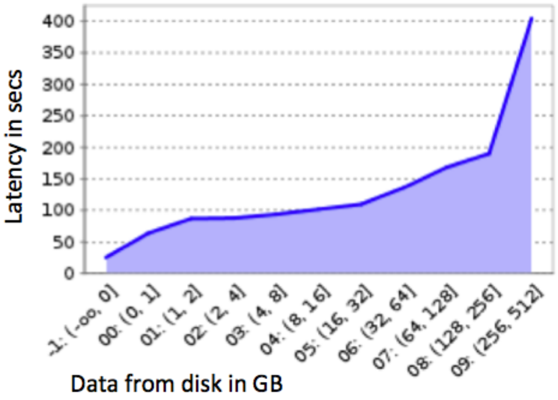
	Savings (per step)
Basic (compared to Dremel)	-11% – 34%
Chunks (partitioning)	-16% – 0%
Optimized storage of int ids	11% – 99.5%
Optimized dictionaries (trie)	0% – 78%
Snappy -- generic compression alg	29% – 49%
Reorder	16% – 55%

Performance

- 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
  - Average response time low # of seconds
  - 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)



Outlook

Next big topics for our team

- Moving beyond AdSpam
- Fast-approximations
    - What is possible if we trade-off speed for accuracy?
      going beyond simple approximations
    - On-going collaboration with visiting researcher Reimar Hofmann
      professor at Hochschule Karlsruhe