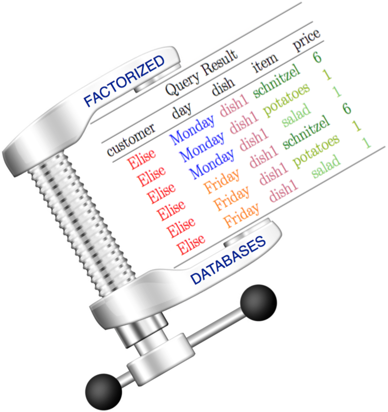
Learning Regression Models over Factorized Joins



Maximilian Schleich

Dan Olteanu

and FDB Team & Collaborators

University of Oxford

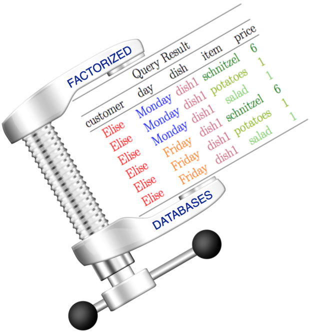
http://www.cs.ox.ac.uk/projects/FDB/

September 2016

Our Key Observations & Results At a Glance

- Join computation entails a high degree of redundancy, which can be avoided by factorized computation and representation.
  - We developed **worst-case optimal factorized join algorithms**. [TODS'15]
  - Factorized joins require **exponentially less time** than standard joins.
  - Aggregates (COUNT, SUM, MIN, MAX) can be computed in **one pass** over factorized data. [VLDB'13]
- Regression models can be learned in **linear time over factorized joins**.
  - This translates to **orders of magnitude performance improvements** over state of the art on real datasets.

Outline



What are Factorized Databases?

Factorizing the Data

Factorizing the Computation

Linear Regression in more Detail

Factorized Databases by Example

Orders (O for short)

customer	day	dish
Elise	Monday	burger
Elise	Friday	burger
Steve	Friday	hotdog
Joe	Friday	hotdog

Dish (D for short)

dish	item
burger	patty
burger	onion
burger	bun
hotdog	bun
hotdog	onion
hotdog	sausage

Items (I for short)

item	price
patty	6
onion	2
bun	2
sausage	4

Consider the natural join of the above relations:

O(customer, day, dish), D(dish, item), I(item, price)

customer	day	dish	item	price
Elise	Monday	burger	patty	6
Elise	Monday	burger	onion	2
Elise	Monday	burger	bun	2
Elise	Friday	burger	patty	6
Elise	Friday	burger	onion	2
Elise	Friday	burger	bun	2
...	...	...	...	...

Factorized Databases by Example

O(customer, day, dish), D(dish, item), I(item, price)

customer	day	dish	item	price
Elise	Monday	burger	patty	6
Elise	Monday	burger	onion	2
Elise	Monday	burger	bun	2
Elise	Friday	burger	patty	6
Elise	Friday	burger	onion	2
Elise	Friday	burger	bun	2
...	...	...	...	...

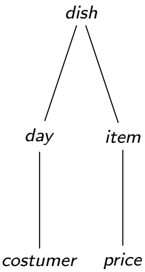
A *flat* relational algebra expression encoding the above query result is:

$$
\begin{array}{lllllllllll}
\langle Elise\rangle & \times & \langle Monday\rangle & \times & \langle burger\rangle & \times & \langle \text{patty}\rangle & \times & \langle 6\rangle & \cup \\
\langle Elise\rangle & \times & \langle Monday\rangle & \times & \langle burger\rangle & \times & \langle \text{onion}\rangle & \times & \langle 2\rangle & \cup \\
\langle Elise\rangle & \times & \langle Monday\rangle & \times & \langle burger\rangle & \times & \langle \text{bun}\rangle & \times & \langle 2\rangle & \cup \\
\langle Elise\rangle & \times & \langle Friday\rangle & \times & \langle burger\rangle & \times & \langle \text{patty}\rangle & \times & \langle 6\rangle & \cup \\
\langle Elise\rangle & \times & \langle Friday\rangle & \times & \langle burger\rangle & \times & \langle \text{onion}\rangle & \times & \langle 2\rangle & \cup \\
\langle Elise\rangle & \times & \langle Friday\rangle & \times & \langle burger\rangle & \times & \langle \text{bun}\rangle & \times & \langle 2\rangle & \cup \ldots
\end{array}
$$

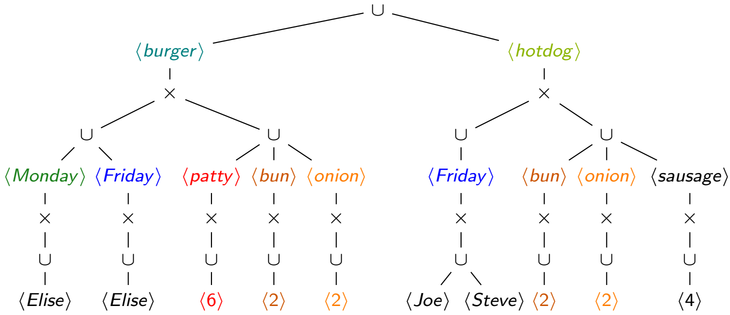
It uses relational product ($\times$), union ($\cup$), and data (singleton relations).

- The attribute names are not shown to avoid clutter.

This is How Factorized Databases Look Like!



Join tree

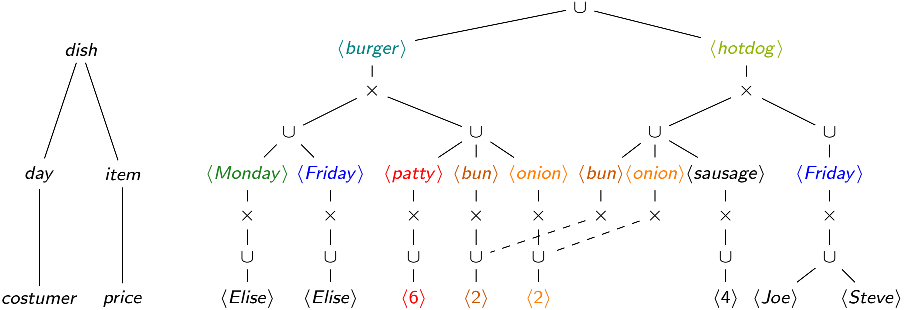


Factorized representation of the join result

There are several *algebraically equivalent* factorized representations defined:

- by distributivity of product over union and their commutativity;
- as groundings of join trees.

.. Now with Further Compression

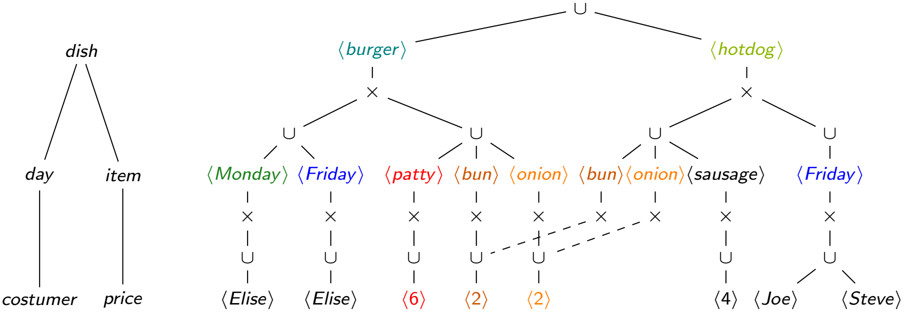


Observation:

- `price` is under `item`, which is under `dish`, but only *depends* on `item`,
- .. so the same `price` appears under an `item` *regardless* of the `dish`.

Idea: *Cache* `price` for a specific `item` and avoid repetition!

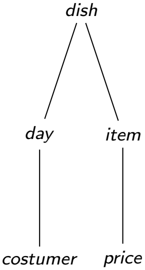
Aggregates over Factorized Databases (1/2)

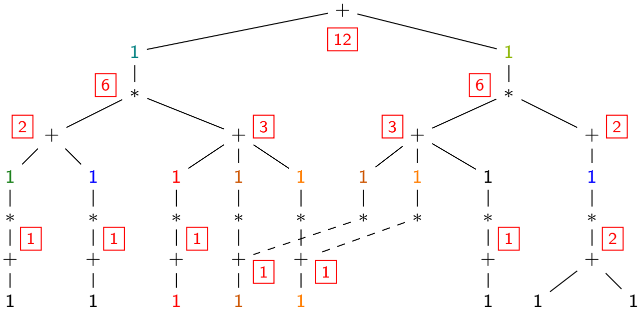


SQL aggregates can be computed in one pass over the factorization:

- `COUNT(*)`:
  - values $\mapsto 1$,
  - $\cup \mapsto +$,
  - $\times \mapsto *$.

Aggregates over Factorized Databases (1/2)

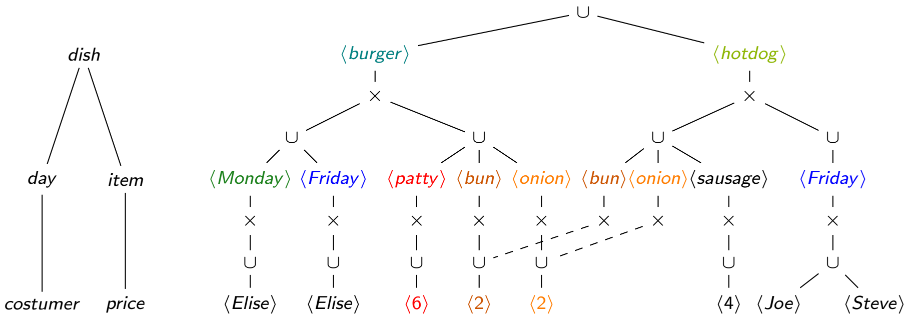




SQL aggregates can be computed in one pass over the factorization:

- COUNT(*):
  - values $\mapsto$ 1,
  - $\cup \mapsto +$,
  - $\times \mapsto *$.

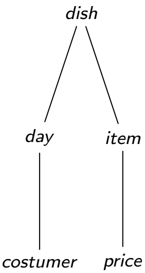
Aggregates over Factorized Databases (2/2)

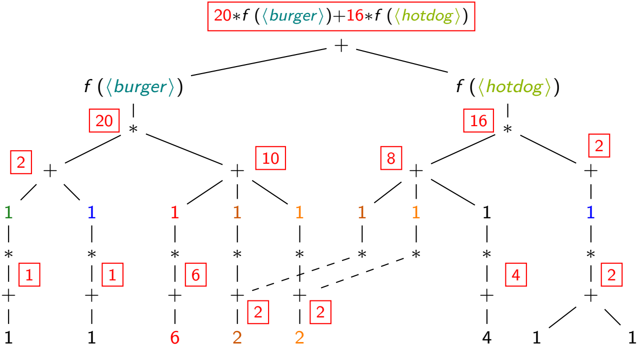


SQL aggregates can be computed in one pass over the factorization:

- `SUM(dish * price)`:
  - Assume there is a function f that turns `dish` into reals.
  - All values except for `dish` & `price` $\mapsto 1$,
  - $\cup \mapsto +$,
  - $\times \mapsto *$.

Aggregates over Factorized Databases (2/2)

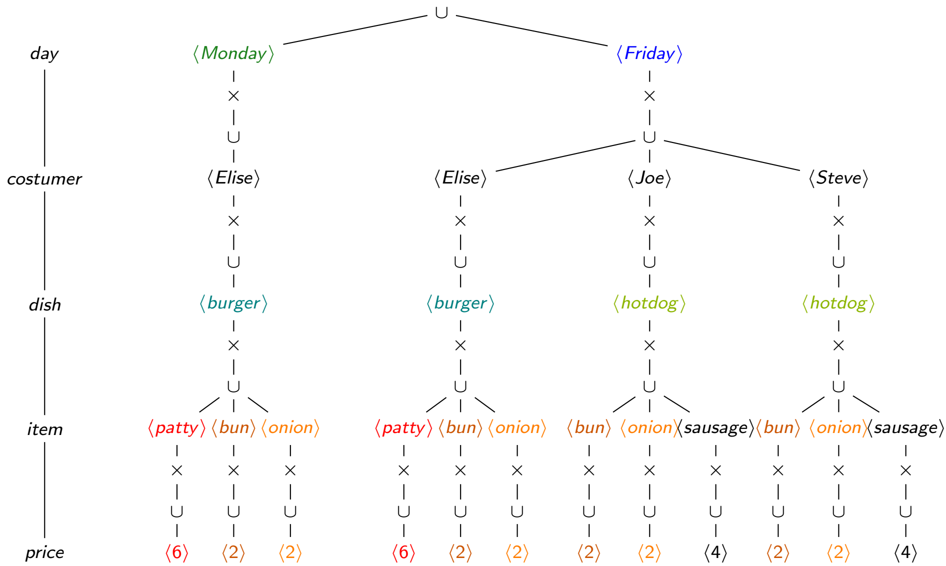




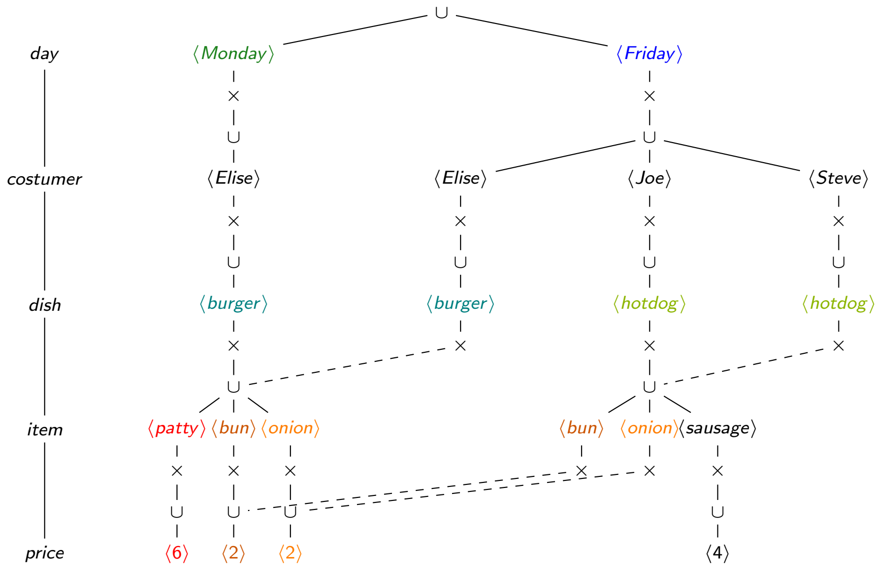
SQL aggregates can be computed in one pass over the factorization:

- SUM(dish * price):
  - Assume there is a function f that turns dish into reals.
  - All values except for dish & price $\mapsto 1$,
  - $\cup \mapsto +$,
  - $\times \mapsto *$.

Just 'Cause We Can: Same Data, Different Factorization

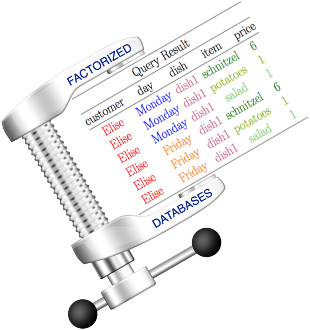


.. and Further Compressed



Which factorized representations should we choose?

Outline



What are Factorized Databases?

Factorizing the Data

Factorizing the Computation

Linear Regression in more Detail

Size of Factorized Databases

The *size* of a factorization is the number of its values.

Example:

$$
\begin{aligned}
F_1 = & \big(\langle 1 \rangle \cup \cdots \cup \langle n \rangle\big) \times \big(\langle 1 \rangle \cup \cdots \cup \langle m \rangle\big) \\
F_2 = & \langle 1 \rangle \times \langle 1 \rangle \cup \cdots \cup \langle 1 \rangle \times \langle m \rangle \\
& \cup \cdots \cup \\
& \langle n \rangle \times \langle 1 \rangle \cup \cdots \cup \langle n \rangle \times \langle m \rangle.
\end{aligned}
$$

- F_1 is factorized, F_2 is flat
- $F_1 \equiv F_2$
- **BUT** $|F_1| = m + n \ll |F_2| = m * n$.

How much space does factorization save?

Size Bounds for Flat and Factorized Join Results

Given a join query Q, for any database $\mathbf{D}$, the join result $Q(\mathbf{D})$ admits

- a flat representation of size $O(|\mathbf{D}|^{\rho^*(Q)})$. [AGM'08]

Size Bounds for Flat and Factorized Join Results

Given a join query Q, for any database $\mathbf{D}$, the join result $Q(\mathbf{D})$ admits

- a flat representation of size $O(|\mathbf{D}|^{\rho^*(Q)})$. [AGM'08]

- a factorization *without caching* of size $O(|\mathbf{D}|^{s(Q)})$. [OZ'11]

Size Bounds for Flat and Factorized Join Results

Given a join query Q, for any database $\mathbf{D}$, the join result $Q(\mathbf{D})$ admits

- a flat representation of size $O(|\mathbf{D}|^{\rho^*(Q)})$. [AGM'08]
- a factorization *without caching* of size $O(|\mathbf{D}|^{s(Q)})$. [OZ'11]
- a factorization *with caching* of size $O(|\mathbf{D}|^{fhtw(Q)})$. [OZ'15]

Size Bounds for Flat and Factorized Join Results

Given a join query Q, for any database $\mathbf{D}$, the join result $Q(\mathbf{D})$ admits

- a flat representation of size $O(|\mathbf{D}|^{\rho^*(Q)})$. [AGM'08]
- a factorization *without caching* of size $O(|\mathbf{D}|^{s(Q)})$. [OZ'11]
- a factorization *with caching* of size $O(|\mathbf{D}|^{fhtw(Q)})$. [OZ'15]

$$\boxed{1 \leq fhtw(Q) \underbrace{\leq}_{\text{up to } \log|Q|} s(Q) \underbrace{\leq}_{\text{up to } |Q|} \rho^*(Q) \leq |Q|}$$

- $|Q|$ is the number of relations in Q
- $\rho^*(Q)$ is the fractional edge cover number of Q
- $s(Q)$ is the factorization width of Q
- $fhtw(Q)$ is the fractional hypertree width of Q

Size Bounds for Flat and Factorized Join Results

Given a join query Q, for any database $\mathbf{D}$, the join result $Q(\mathbf{D})$ admits

- a flat representation of size $O(|\mathbf{D}|^{\rho^*(Q)})$. [AGM'08]
- a factorization *without caching* of size $O(|\mathbf{D}|^{s(Q)})$. [OZ'11]
- a factorization *with caching* of size $O(|\mathbf{D}|^{fhtw(Q)})$. [OZ'15]

These size bounds are asymptotically tight!

- **Best possible bounds** for representations obtained by grounding join trees of Q, *but not necessarily <u>instance</u> optimal*:

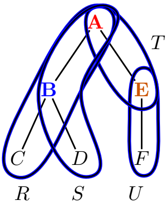
  There exists databases $\mathbf{D}$ such that the grounding of any join tree of Q over $\mathbf{D}$ has sizes: $\Omega(|\mathbf{D}|^{\rho^*(Q)})$, $\Omega(|\mathbf{D}|^{s(Q)})$, and respectively $\Omega(|\mathbf{D}|^{fhtw(Q)})$.

Factorization Example

Consider the following join query:

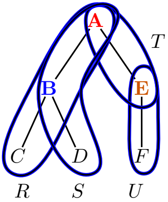
$$Q = R(\mathbf{A}, \mathbf{B}, C), S(\mathbf{A}, \mathbf{B}, D), T(\mathbf{A}, \mathbf{E}), U(\mathbf{E}, F).$$

Its hypergraph (relations = hyperedges, variables = nodes) and join tree:



We assume for simplicity databases **D** such that $|R| = |S| = |T| = |U| = O(|\mathbf{D}|)$.

Fractional Edge Cover Number $\rho^*(Q)$



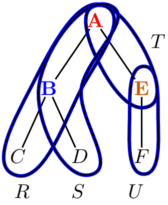
- Upper bound $O(|\mathbf{D}|^3)$ on the size of query result:
  Edges R, S, U cover the whole query: $\mathrm{EdgeCover}(Q) \leq 3$.

- Lower bound $\Omega(|\mathbf{D}|^3)$ on the size of query result:
  Each of C, D, and F must be covered by an edge: $\mathrm{IndSet}(Q) \geq 3$.

  $\Rightarrow \rho^*(Q) = 3$

  $\Rightarrow$ The size of the query result is at most cubic and
  there are databases for which the size must be cubic.

Factorization Width $s(Q)$



$$\bigcup_{a \in \mathbf{A}} \left(\langle a \rangle \times \bigcup_{b \in \mathbf{B}} \left(\langle b \rangle \times \left(\bigcup_{c \in C} \langle c \rangle \right) \times \left(\bigcup_{d \in D} \langle d \rangle \right) \right) \times \bigcup_{e \in \mathbf{E}} \left(\langle e \rangle \times \left(\bigcup_{f \in F} \langle f \rangle \right) \right) \right)$$

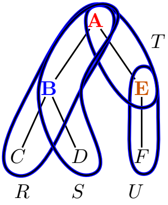
The number of values for a variable is dictated by the number of actual combinations of values for its ancestors:

- One value $\langle f \rangle$ for each tuple (a, e, f) in the query result.
- The number of F-values is $|\pi_{A,E,F}(Q(\mathbf{D}))|$.

Size of factorization = sum of sizes of results of **subqueries along paths**.

- s(Q) is the largest $\rho^*(Q')$ for subqueries Q' along paths in Q.

Factorization Width $s(Q)$



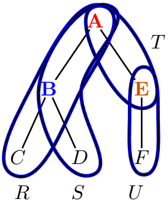
- Path A–$\mathbf{E}$–F has $\rho^* = 2$.
  $\Rightarrow$ The number of F-values is $\leq |\mathbf{D}|^2$, but can be $\sim |\mathbf{D}|^2$.
- All other root-to-leaf paths have $\rho^* = 1$.
  $\Rightarrow$ The number of values for any other variable is $\leq |\mathbf{D}|$.

$s(Q) = 2$ $\Rightarrow$ Factorization size $\sim |\mathbf{D}|^2$

Recall that $\rho^*(Q) = 3$ $\Rightarrow$ Flat size $\sim |\mathbf{D}|^3$

Fractional Hypertree Width $fhtw(Q)$

Idea: Avoid repeating an identical expression and cache it instead.



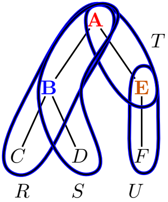
$$\bigcup_{a \in \mathbf{A}} \left[\langle a \rangle \times \cdots \times \bigcup_{e \in \mathbf{E}} \left(\langle e \rangle \times \left(\bigcup_{f \in F} \langle f \rangle \right) \right) \right]$$

- F only depends on **E** and not on **A**.
- A value $\langle e \rangle$ binds with the same union $\bigcup_{(e,f) \in U} \langle f \rangle$ regardless of the value $\langle a \rangle$ above it.

  $\Rightarrow$ Define $U_e = \bigcup_{(e,f) \in U} \langle f \rangle$ for each value $\langle e \rangle$ and use U_e instead of the union $\bigcup_{(e,f) \in U} \langle f \rangle$.

Fractional Hypertree Width $fhtw(Q)$

Idea: Avoid repeating an identical expression and cache it instead.



A factorization with caching would be:

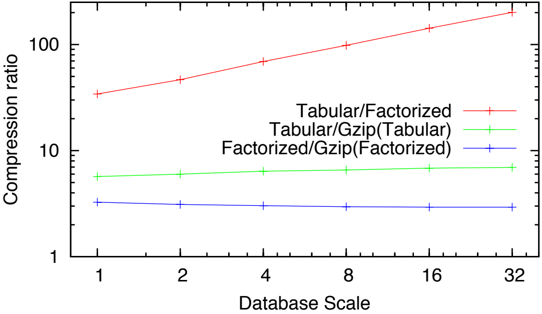
$$\bigcup_{a \in \mathbf{A}} \left[\langle a \rangle \times \cdots \times \bigcup_{e \in \mathbf{E}} \left(\langle e \rangle \times U_e \right) \right]; \qquad \left\{ U_e = \bigcup_{(e,f) \in U} \langle f \rangle \right\}$$

The width $fhtw(Q)$:

- Like $s(Q)$, it is the largest $\rho^*(Q')$ for subqueries Q' along paths in Q,
- **BUT** for each variable, only consider those ancestors it depends on!
  For F, we only consider the subquery over E and F (i.e., U) and ignore A.

For our example: $fhtw(Q) = 1 < s(Q) = 2 < \rho^*(Q) = 3$.

Compression Contest: Factorized vs. Zipped Relations



Setup: [BKOZ'13]

- Flat = flat result of join $\text{Orders} \bowtie \text{Dish} \bowtie \text{Items}$ in CSV text format
- Gzip (compression level 6) outputs binary format
- Fatorized output in text format (each digit takes one character)

Observations:

- Gzip does not exploit distant repetitions!
- Factorizations can be arbitrarily more succinct than gzipped relations.
- Gzipping factorizations improves the compression by 3x.

Factorization Gains in Practice (1/3)

US retailer dataset used for LogicBlox/Predictix analytics

- Relations: Inventory (84M), Sales (1.5M), Clearance (368K), Promotions (183K), Census (1K), Location (1K).
- Compression factors (caching not used):
  - **26.61x** for natural join of Inventory, Census, Location.
  - **159.59x** for natural join of Inventory, Sales, Clearance, Promotions

Factorization Gains in Practice (2/3)

LastFM public dataset

- Relations: UserArtists (93K), UserFriends (25K), TaggedArtists (186K).
- Compression factors:
  - **143.54x** for joining two copies of Userartists and Userfriends

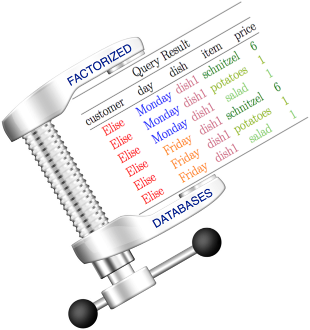
    With caching: **982.86x**
  - **253.34x** when also joining on TaggedArtists
  - **2.53x/ 3.04x/ 924.46x** for triangle/4-clique/bowtie query on UserFriends
  - **9213.51x/ 552Kx/ $\geq$86Mx** for versions of triangle/4-clique/bowtie queries with copies for UserArtists for each UserFriend copy

Factorization Gains in Practice (3/3)

Twitter public dataset

- Relation: Follower-Followee (1M)
- Compression factors:
  - **2.69x** for triangle query
  - **3.48x** for 4-clique query
  - **4918.73x** for bowtie query

Outline



What are Factorized Databases?

Factorizing the Data

Factorizing the Computation

Linear Regression in more Detail

Join Queries

Given a join query Q, for any database $\mathbf{D}$, the join result $Q(\mathbf{D})$ can be computed in

- $O(|\mathbf{D}|^{\rho^*(Q)})$ as flat representation [NPRR'12]
- $O(|\mathbf{D}|^{s(Q)})$ as factorization *without caching* [OZ'15]
- $O(|\mathbf{D}|^{fhtw(Q)})$ as factorization *with caching* [OZ'15]

The above times essentially follow the succinctness gap. They are:

- worst-case optimal within the given representation model.
- modulo poly-log factors in $|\mathbf{D}|$.
- with respect to *data* complexity.

Aggregates & Regression Models

SQL aggregates can be computed in one pass over factorized data. [BKOZ'13]

Polynomial Regression and Factorization Machines models of degree d can be learned over a factorized relation with schema $(X_1, \ldots, X_n)$ in two steps:

[OS'16]

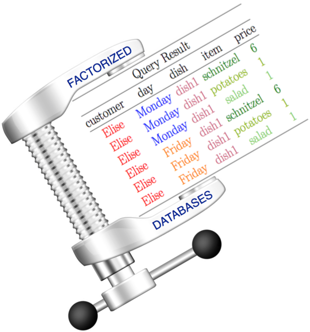
1. Data-dependent step: Aggregate computation

$$\sum \times_{i \in S} X_i, \text{ where } S \subseteq \{X_1, \ldots, X_n\} \text{ is a multiset of arity } \leq 2d.$$

2. Data-independent step: Convergence of the model parameter

   Perform fixpoint computation on top of the aggregates.

Outline



What are Factorized Databases?

Factorizing the Data

Factorizing the Computation

Linear Regression in more Detail

Regression Recap

- Training dataset computed as join of database tables

$$\begin{pmatrix} y^{(1)} & x_1^{(1)} & \dots & x_n^{(1)} \\ y^{(2)} & x_1^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ y^{(m)} & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix}$$

  $y^{(i)}$ are labels, $x_1^{(i)}, \dots, x_n^{(i)}$ are features, all mapped to reals.

- We'd like to learn the parameters $\Theta = (\theta_0, \dots, \theta_n)$ of the *linear* function

$$h_\Theta(x) = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n.$$

  For uniformity, we add $x_0 = 1$ so that $h_\Theta(x) = \sum_{k=0}^{n} \theta_k x_k$.

- Function h_Θ approximates the label y of unseen tuples $(x_1, \dots, x_n)$.

Least-Squares Linear Regression

- We consider the least squares regression model with the cost function:

$$J(\Theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2$$

Least-Squares Linear Regression

- We consider the least squares regression model with the cost function:

$$J(\Theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2$$

Batch gradient descent (BGD):

- Repeatedly change Θ to make $J(\Theta)$ smaller until convergence:

$$\begin{aligned}\forall 0 \leq j \leq n : \theta_j &:= \theta_j - \alpha \frac{\delta}{\delta\theta_j} J(\Theta) \\ &:= \theta_j - \alpha \sum_{i=1}^{m}(\sum_{k=0}^{n} \theta_k x_k^{(i)} - y^{(i)}) x_j^{(i)}.\end{aligned}$$

- α is the learning rate.

Least-Squares Linear Regression

- We consider the least squares regression model with the cost function:

$$J(\Theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2$$

Batch gradient descent (BGD):

- Repeatedly change Θ to make $J(\Theta)$ smaller until convergence:

$$\begin{aligned}\forall 0 \le j \le n : \theta_j &:= \theta_j - \alpha \frac{\delta}{\delta\theta_j} J(\Theta)\\ &:= \theta_j - \alpha \sum_{i=1}^{m}(\sum_{k=0}^{n}\theta_k x_k^{(i)} - y^{(i)})x_j^{(i)}.\end{aligned}$$

- α is the learning rate.
- We consider wlog that y is also part of x's and has $\theta = -1$.
- We thus need to compute the following aggregates:

$$\forall 0 \le j \le n : S_j = \sum_{i=1}^{m}(\sum_{k=0}^{n}\theta_k x_k^{(i)})x_j^{(i)}.$$

F: Factorised Regression

- The sums

$$\forall 0 \leq j \leq n : S_j = \sum_{i=1}^{m} (\sum_{k=0}^{n} \theta_k x_k^{(i)}) x_j^{(i)}.$$

can be rewritten so that we can express the cofactor of each θ_k in S_j:

$$\forall 0 \leq j \leq n : S_j = \sum_{k=0}^{n} \theta_k \times \text{Cofactor}_{kj}$$

where $\text{Cofactor}_{kj} = \sum_{i=1}^{m} x_k^{(i)} x_j^{(i)}$

- We decouple the computation of cofactors from convergence of Θ.
  - The cofactor computation only depends on the input data.
  - Convergence can be done once the cofactors are computed.

- **F** computes the cofactors **in one pass** over the factorised input dataset.
  - The redundancy in the flat data is not necessary for learning!

Complexity of F

For a training dataset defined by a join query Q over any database $\mathbf{D}$, $\mathbf{F}$ learns the parameters of any linear function in time $O(|\mathbf{D}|^{\mathit{fhtw}(Q)})$.

For (α-)acyclic joins, $\mathit{fhtw} = 1$ and $\mathbf{F}$ learns in optimal time.

Extensions of **F**

- Push cofactor matrix computation inside the factorized join computation!
  - Removing the lion's share of the computation, and computing cofactor matrix in one pass over the input data.

- **F/SQL**: Compute cofactor matrix in SQL.
  - Allowing for direct implementation of **F** in any standard Relational DBMS.

- **F** currently supports
  - any arbitrary nonlinear basis functions,
  - polynomial regression models, and
  - factorisation machines.

  The *data* complexity stays the same as for linear regression.

- Multi-core and distributed learning further improve performance of joins and aggregates.

- Categorical features needed in real-world cases.
  - Resulting large number of features require a slightly different approach.

Learning Regression Models in Practice

Competing systems:

- **F**: Our learner over factorized joins
  - Next slide: Times for running in one thread on one machine.
- **R** (QR-decomposition)
- **P**ython StatsModels (ordinary least squares)
- and **M**ADlib (generalized linear model (glm), ordinary least squares (ols))

Datasets:

- **US Retailer**: Predict the amount of inventory units.
- **LastFM**: Predict how often a user would listen to an artist based on similar information for its friends.

F versus R, Python StatsModels and MADlib

		US retailer	US retailer	LastFM	LastFM
model degree/# params/#agg		1/31/496	2/496/123256	1/10/55	2/55/1540
Size	Factorized	97,134,675	97,134,675	315,818	315,818
	Flat	2,585,046,352	2,585,046,352	590,793,800	590,793,800
	Compression	26.61×	26.61×	982.86×	982.86×
Join	PostgreSQL	249.41	249.41	61.33	61.33
	F	3.28	3.28	0.065	0.065
Import Time	R	1189.12*	1189.12*	155.91	276.77
	P	1164.40*	1164.40*	179.16	328.97
Learn Time	M (glm)	2671.88	2937.49	572.88	746.50
	R	810.66*	873.14*	268.04	466.52
	P	1199.50*	1277.10*	35.74	148.84
Total Time	**F**	4.206	30.02	0.081	0.247
	M (ols)	680.60	3186.90	196.60	1382.49
	M (glm)	2921.29*	–	807.83	–
	R	2249.19*	–	804.62	–
	P	2613.31*	–	539.14	–

- We consider Polynomial Regression models of degrees 1 and 2.
- Performance numbers are in seconds.
- We assume data is in memory and sorted.
  - P and R have an extra DBMS export & import step (shown explicitly).

Thank you!