

relationalAI

A complex, abstract network graph serves as the background for the slide. It consists of numerous small, semi-transparent blue dots connected by thin, light-blue lines, forming a dense web of triangles and polygons. This pattern repeats across the entire slide, creating a sense of depth and connectivity.

Molham Aref

Reinventing the Database for AI

July 19, 2019

We are a mission-based team

Scientific Impact

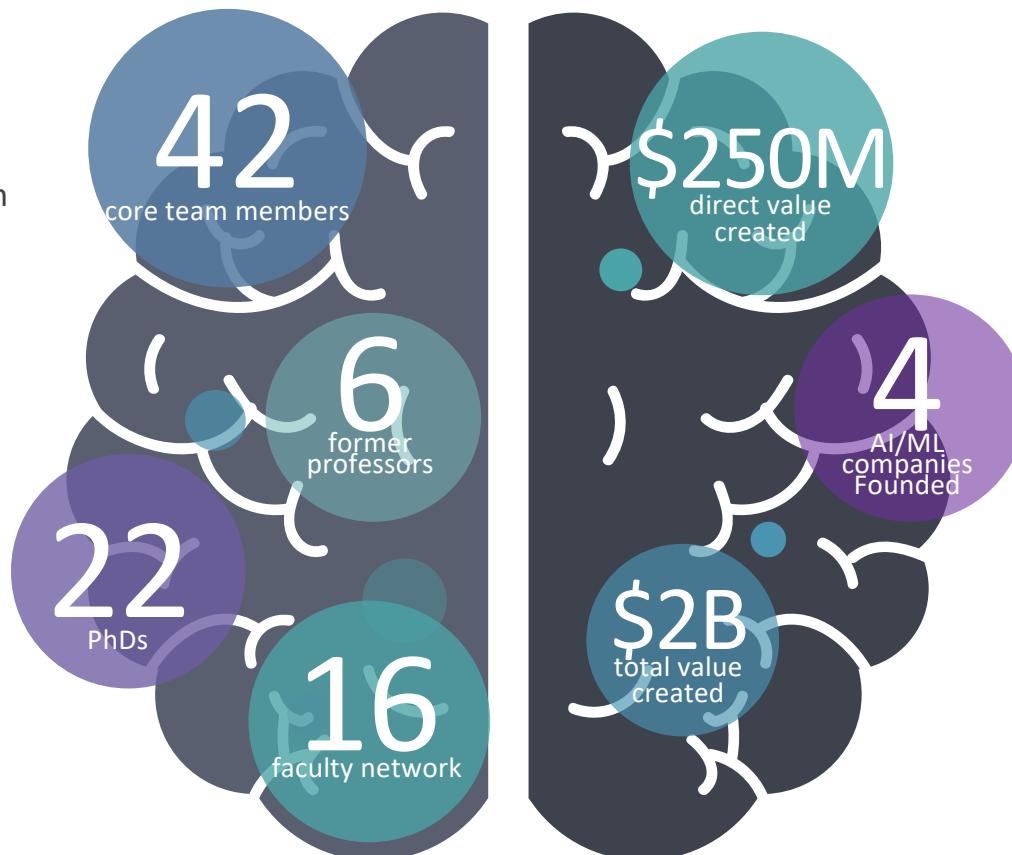
Deep computer science and mathematical expertise from several technical communities:

- Database systems and theory
- Machine learning
- Programming languages
- Operations research

2K+ publications

90K+ citations
(35K+ in last 5 years)

37+ award-winning papers
(3 this year!)



AI and ML Industrial Impact



relationalAI

The Case for Relational Artificial Intelligence

A New Technology Category

What if I tell you

Databases should be Relational

Not Controversial but it used to be

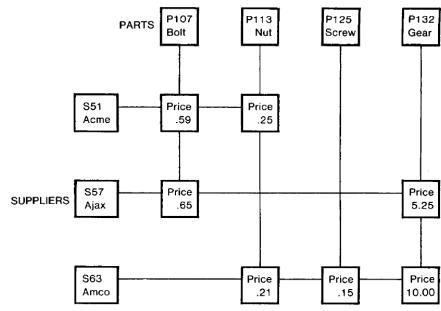
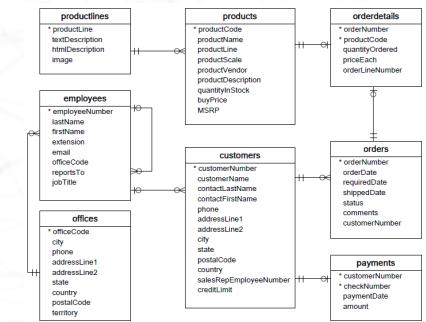


Fig. 1(a). A "Navigational" Database.

Navigational vs Relational



In the Navigational vs Relational DB wars of the 1980's,
Navigational DB's were the incumbent and Relational DBs were the underdog!

relationalAI

The Great Debate

• database



Navigational



Charles Bachman

Weighing in with:

- Turing Award for Databases
- Integrated Data Store (IDS)
- Illustrious career at GE and Honeywell

Argument:

- Performance
(it's impossible to implement the relational model efficiently)
- Programmers won't get it
(Cobol programmers can't possibly understand relational languages)

1974

Project

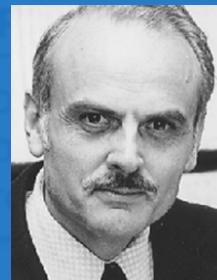
name

startDate

endDate

departmentID (FK)

Relational



Ted Codd

Weighing in with:

- Researcher at IBM

Argument:

- Separation of the What from the How (Argument for declarativity)
- Domain experts will get it (and they are cheaper and more plentiful than programmers)

Department

name

departmentID (FK)

personID (FK)

Entity

ID (PK)

name

1974

Navigational



Charles Bachman

Weighing in with:

- Turing Award for Databases
- Integrated Data Store (IDS)
- Illustrious career at Endeca/Moneyline

Argument:

- Performance
(it's impossible to implement the relational model efficiently)
- Programmers won't get it
(Cobol programmers can't possibly understand relational languages)

Relational



Ted Codd

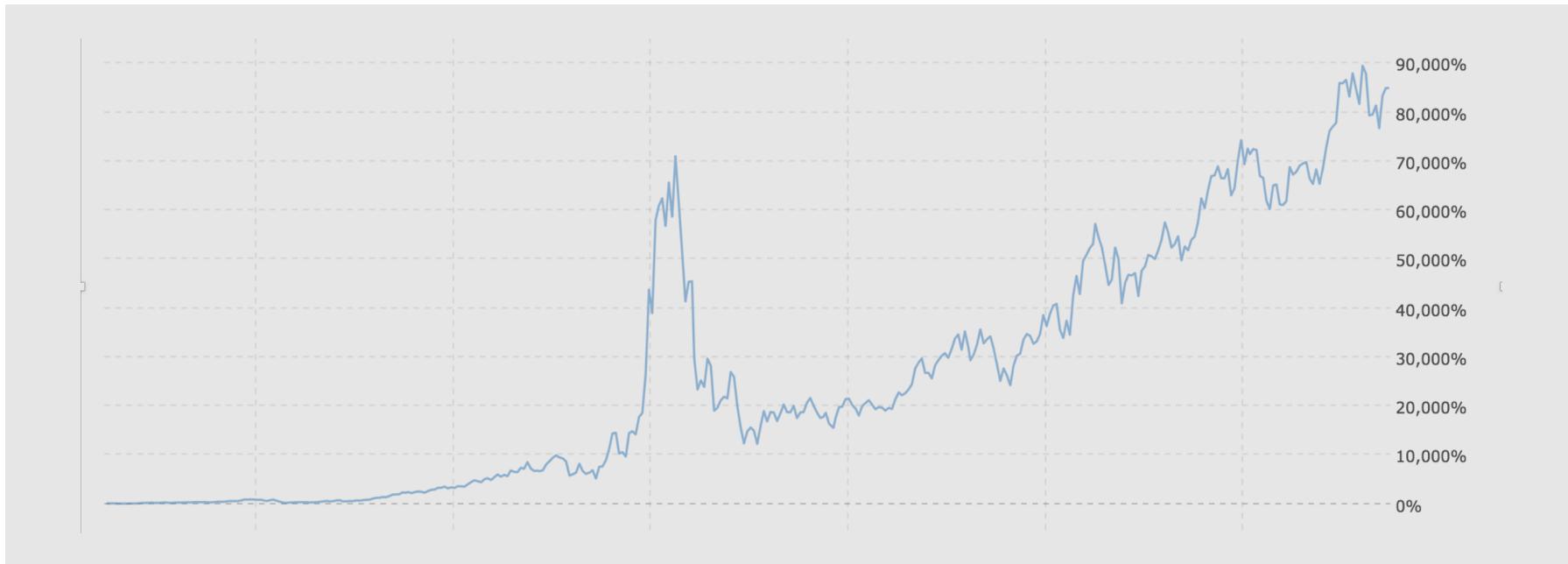
Weighing in with:

- Researcher at IBM

Argument:

- Separation of the What from the How (Argument for declarativity)
- Domain experts will get it
(and they are cheaper and more plentiful than programmers)

SO WHO WON?



ORACLE®

Oracle (formerly Relational Software, Inc.)

- Launched RDBMS in 1979
- IPO in 1986
- Current Market Cap: **\$190.6B**

INGRES 2,000,000 Shares

Relational Technology, Inc.

Common Stock

The executive officers and directors of the Company and their ages as of March 31, 1988 are as follows:

Name	Age	Position
Gary J. Morgenthaler	39	Chairman of the Board, Chief Executive Officer and Director
Paul E. Newton	44	President, Chief Operating Officer and Director
Nicholas Birtles	43	Vice President, International Operations
Robert Healy	45	Vice President, Marketing
Lawrence A. Rowe	39	Vice President, Advanced Development
P. Michael Seashols	42	Vice President, Sales and Marketing
William M. Smartt	45	Vice President, Finance and Administration and Chief Financial Officer
Martin J. Sprinzen	40	Vice President, Engineering
Eugene Wong	53	Secretary
Robert C. Miller(1)	44	Director
Charles G. Moore(1)(2)	44	Director
Michael R. Stonebraker	44	Director
William H. Younger, Jr. (1)(2)	38	Director

Goldman, Sachs & Co. **Robertson, Colman & Stephens**

The date of this Prospectus is May 17, 1988.



Ingres (formerly Relational Technology, Inc.)

- Launched RDBMS in 1981
- IPO'd in 1988 (sold prematurely to ASK in 1989)

RDBMS Popularity

DB-Engines Ranking May 2019

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

Relational DBMS

1. Oracle

Relational DBMS

2. MySQL

Relational DBMS

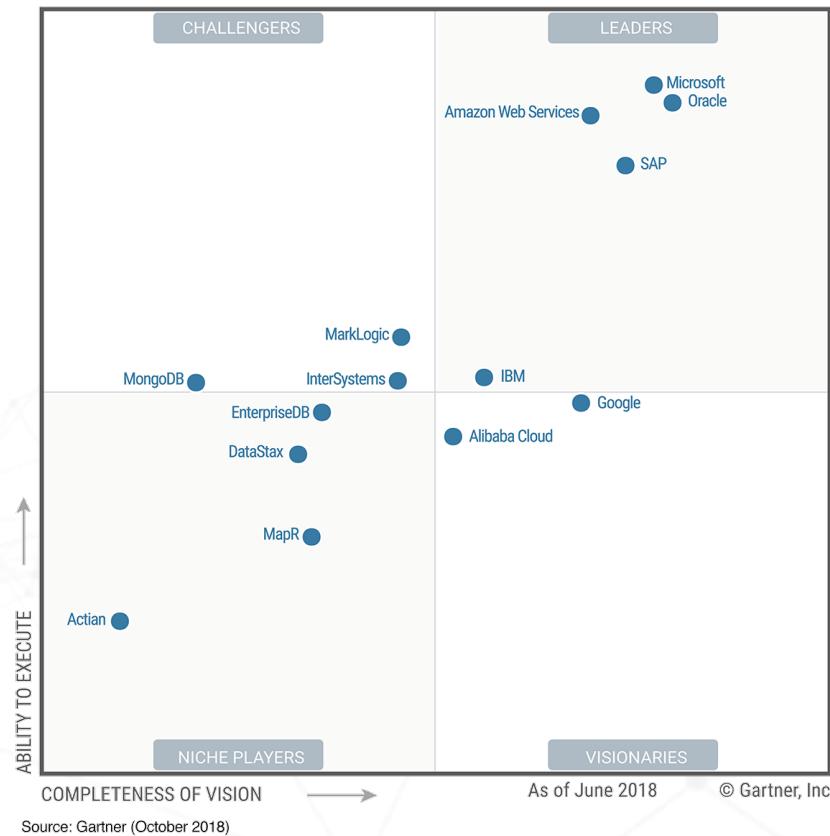
3. Microsoft SQL Server

Relational DBMS

4. PostgreSQL

Analysts agree

Figure 1. Magic Quadrant for Operational Database Management Systems

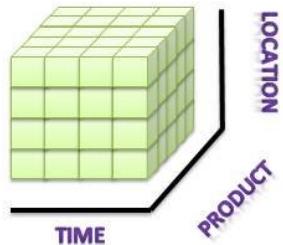


Why?

What if I tell you

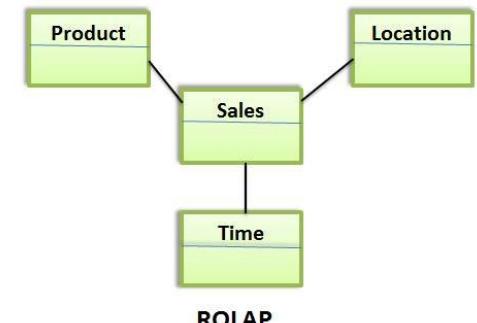
Business Intelligence should be Relational

Not Controversial but it used to be



MOLAP

MOLAP vs ROLAP



In the Multidimensional (i.e. Tensor) **vs** Relational OLAP wars of the 1990's,
MOLAP was the incumbent and ROLAP was the underdog!



Tableau Software

- Launched in 2002
- IPO in 2013
- Current Market Cap: **\$11.6B**

Analysts agree



Why?

What if I tell you

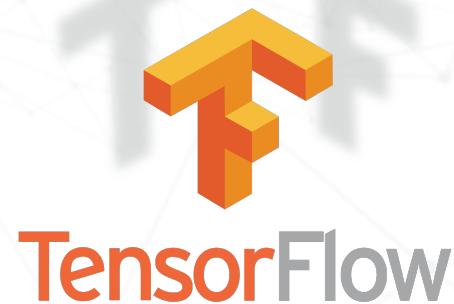
Artificial Intelligence should be Relational

What if I tell you

No way!!

Relational systems are **too slow!**

Tensors and linear algebra are the way we've always done it



I am here to tell you

Relational Artificial Intelligence is Inevitable

relationalAI

Why?

Rest of the talk

The Need for Speed

“We track about **47 different hardware startups** that all have a unique approach” to accelerating AI.

Greg Brockman, CTO OpenAI, interviewed by Reid Hoffman, May 30, 2019

“13 private chip companies focused on the AI market have raised more than \$1.2 billion in venture-capital funding”

- Barron’s article “AI Chip Market Will Soar to \$34 Billion in Five Years”, Feb 20, 2019

“Today the job of training machine learning models is limited by compute, if we had faster processors we’d run bigger models...in practice we train on a reasonable subset of data that can finish in a matter of months. **We could use improvements of several orders of magnitude – 100x or greater.”**

Greg Diamos, Senior Researcher, SAIL, Baidu, From EE Times – September 27, 2016

AI's biggest challenges are computational!

ACCURACY

Search for better

- Parameters
- Hyper parameters
- Features
- Models

Don't make assumptions that you don't need to make (e.g. i.i.d. assumption)

VERSATILITY

Reasoning and (generalized) inference: From observations to unknowns in any time period

Inference of any property in the model (e.g., it's just as easy to infer price from sales as it is to infer sales from price)

ROBUSTNESS

Many "big data" problems are really a big collection of small data problems

Overcome challenges with small, incomplete, and dirty data problems by incorporating prior knowledge and expertise

SELF-SUPERVISION

"The future will be self-supervised" Yann LeCun

Build models of the world by observing it and searching model space for the models that have the most explanatory power

INTERPRETABILITY

Searching for models that are accurate and interpretable is harder than searching for accurate models

Interpretation in terms of prior knowledge and in language & ontology that humans understand

EXPLAINABILITY

Explainability typically implanted via separate shadow models that have to be learned

Explanation in terms of prior knowledge and in language & ontology that humans understand

FAIRNESS

It's not enough to exclude gender, ethnicity, race, age, etc as features to the models. Other features might be correlated.

Prejudice is a computational limitation: Reasoning about each person vs reasoning about the group

CAUSALITY

Understanding causality beyond A/B testing

Computationally very expensive

The Path to Performance: **Brawn**

Constant factors – Do same amount of work faster (i.e., brawn)

- **Latency hiding:** Memory hierarchy and network latencies (e.g., in memory and near-data computing)
- **Parallelization:** SIMD, multi-core, accelerators (e.g., GPU, TPU, FPGA)
- **Specialization:** Specialize for workload (e.g., JIT compilation), specialize for data

The Path to Performance: Brains and Brawn

Asymptotics – Do less work (i.e., brains)

- **Specialize algorithm** by exploiting problem structure
 - Algebraic (e.g., groups, semi rings, rings)
 - Combinatorial (e.g., fractional hypertree width)
 - Statistical (e.g., samples and sketches)
 - Geometric (e.g., fast multipole method)
- **Solve similar** but more tractable problem
 - Approximation (with error bars)

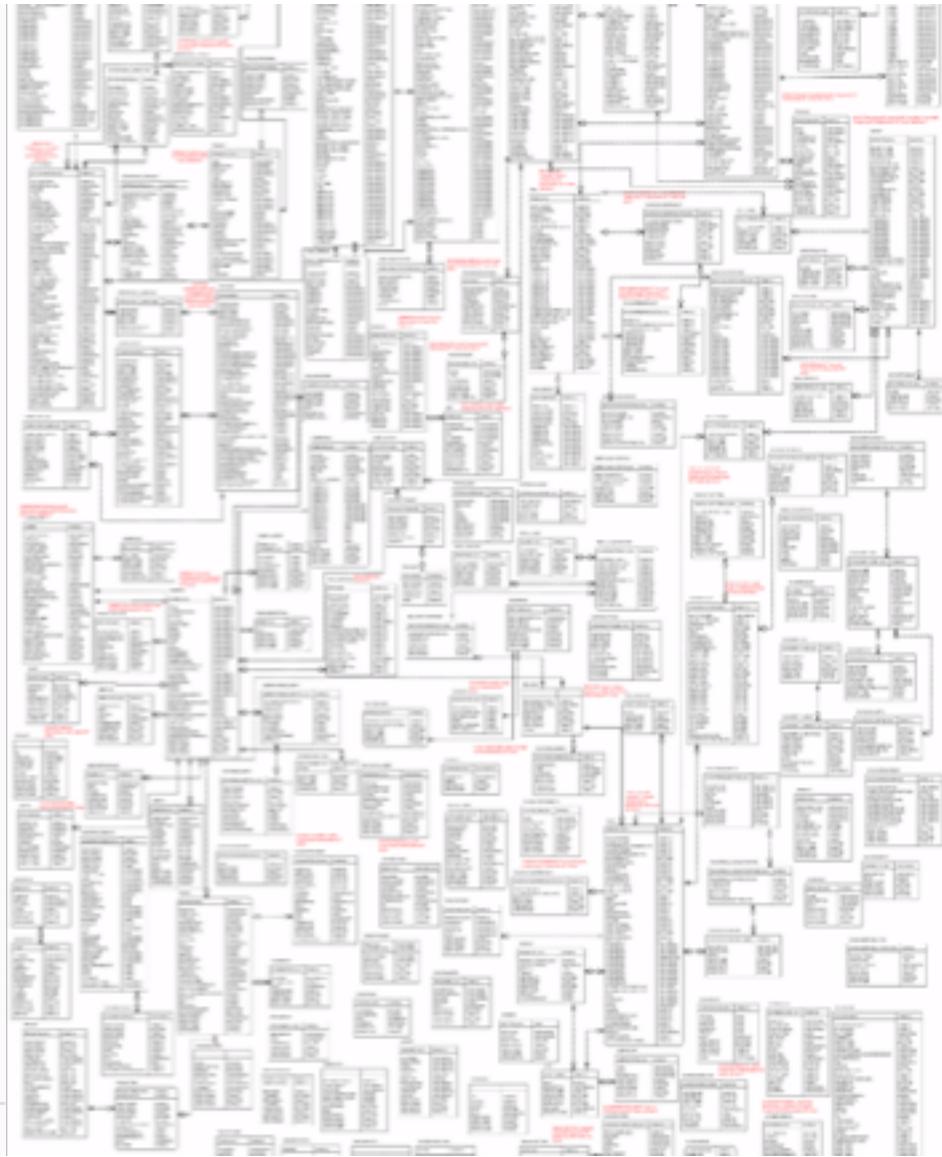
relationalAI

Brains

Do Less Work

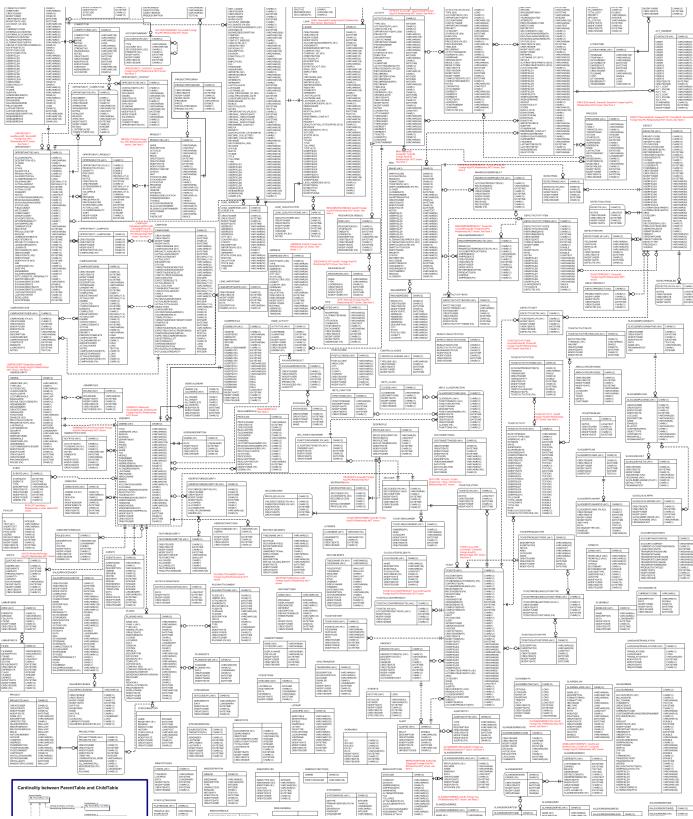
The relational model dominates data management

- The last 40 years have witnessed massive adoption of the relational model
 - It's hard to find any examples today of enterprises whose data isn't in a relational database
- Millions of human hours invested in building relational models and populating them with data
- Relational databases are rich with knowledge of the underlying domains that they model
- The availability and accuracy of large amounts of curated data has made it possible for humans (BI) and machines (AI) to **learn** from the past and to **predict** the future

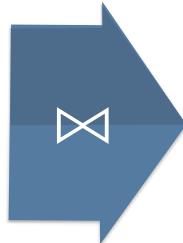




What's the first thing we do when we build predictive models?



Feature extraction query



We work hard to **throw away** all relational structure (and semi-structure) we worked so hard to build

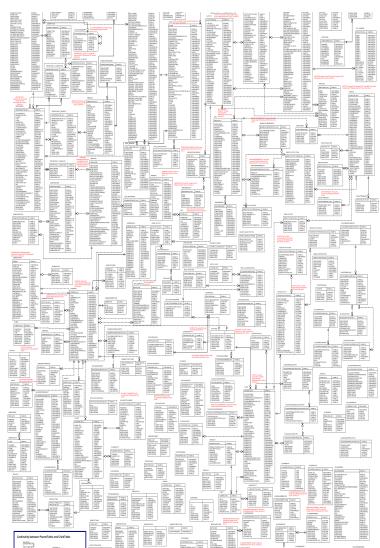
We end up throwing away
important domain
knowledge
that can help us build better
AI models

Examples

Features

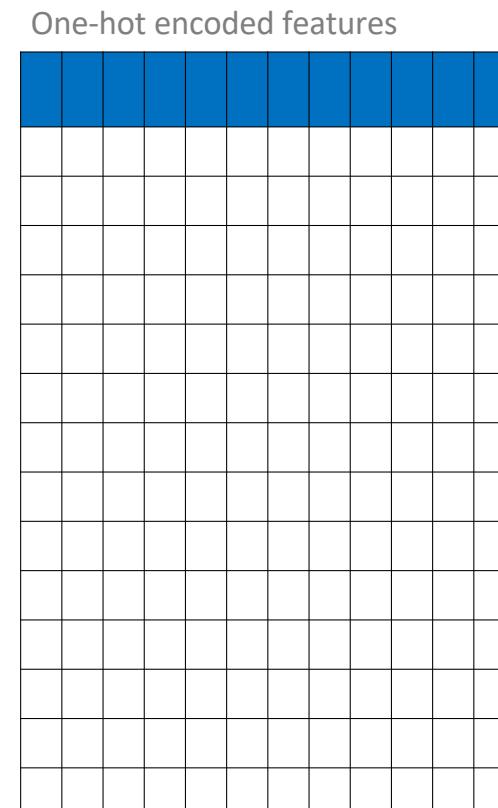


The wastefulness does not end there



Features w/zero filling

Training Samples



MACHINE LEARNING

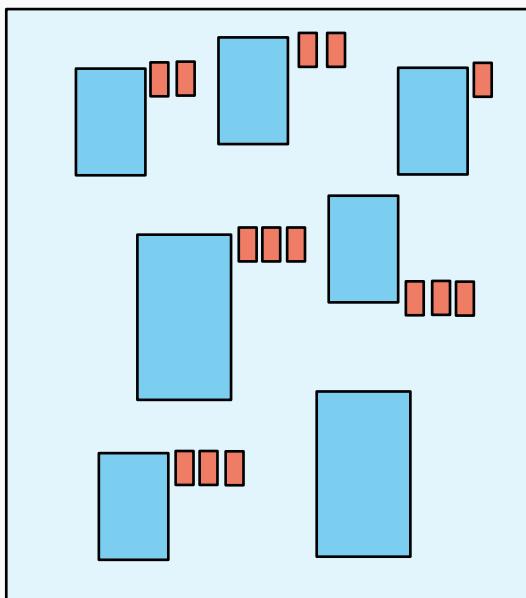
The wastefulness does not end there

Features

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	10010	10011	10012	10013	10014	10015	10016	10017	10018	10019	10020	10021	10022	10023	10024	10025	10026	10027	10028	10029	10030	10031	10032	10033	10034	10035	10036	10037	10038	10039	10040	10041	10042	10043	10044	10045	10046	10047	10048	10049	10050	10051	10052	10053	10054	10055	10056	10057	10058	10059	10060	10061	10062	10063	10064	10065	10066	10067	10068	10069	10070	10071	10072	10073	10074	10075	10076	10077	10078	10079	10080	10081	10082	10083	10084	10085	10086	10087	10088	10089	10090	10091	10092	10093	10094	10095	10096	10097	10098	10099	100100	100101	100102	100103	100104	100105	100106	100107	100108	100109	100110	100111	100112	100113	100114	100115	100116	100117	100118	100119	100120	100121	100122	100123	100124	100125	100126	100127	100128	100129	100130	100131	100132	100133	100134	100135	100136	100137	100138	100139	100140	100141	100142	100143	100144	100145	100146	100147	100148	100149	100150	100151	100152	100153	100154	100155	100156	100157	100158	100159	100160	100161	100162	100163	100164	100165	100166	100167	100168	100169	100170	100171	100172	100173	100174	100175	100176	100177	100178	100179	100180	100181	100182	100183	100184	100185	100186	100187	100188	100189	100190	100191	100192	100193	100194	100195	100196	100197	100198	100199	100200	100201	100202	100203	100204	100205	100206	100207	100208	100209	100210	100211	100212	100213	100214	100215	100216	100217	100218	100219	100220	100221	100222	100223	100224	100225	100226	100227	100228	100229	100230	100231	100232	100233	100234	100235	100236	100237	100238	100239	100240	100241	100242	100243	100244	100245	100246	100247	100248	100249	100250	100251	100252	100253	100254	100255	100256	100257	100258	100259	100260	100261	100262	100263	100264	100265	100266	100267	100268	100269	100270	100271	100272	100273	100274	100275	100276	100277	100278	100279	100280	100281	100282	100283	100284	100285	100286	100287	100288	100289	100290	100291	100292	100293	100294	100295	100296	100297	100298	100299	100300	100301	100302	100303	100304	100305	100306	100307	100308	100309	100310	100311	100312	100313	100314	100315

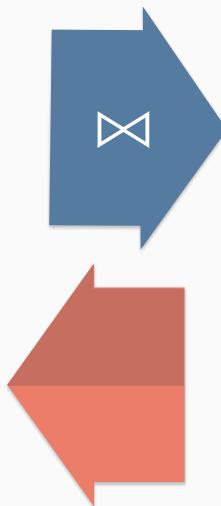
What would a database do?

1. Database



s: Sufficient statistics generated from model spec and feature extraction query.
Computed via aggregations

2. Feature extraction query



Features

3. Model specification (e.g., “degree 2 ridge regression”)

Number of Aggregates Varies By Model Class

■ Supervised

- Regression

Model	# features	# params	# aggregates
Linear regression	n	$n + 1$	$\Theta(n^2)$
Polynomial regression	$\Theta(n^d)$	$\Theta(n^d)$	$\Theta(n^{2d})$
Factorization machines	$\Theta(n^d)$	$\Theta(nr)$	$\Theta(n^{2d})$

n : # input features
 d : degree
 r : rank

- Classification

Model	# features	# aggregates
Decision trees	$\Theta(n)$	$\Theta(nbh)$

b : branching factor, h : depth
(data-dependent)

■ Unsupervised

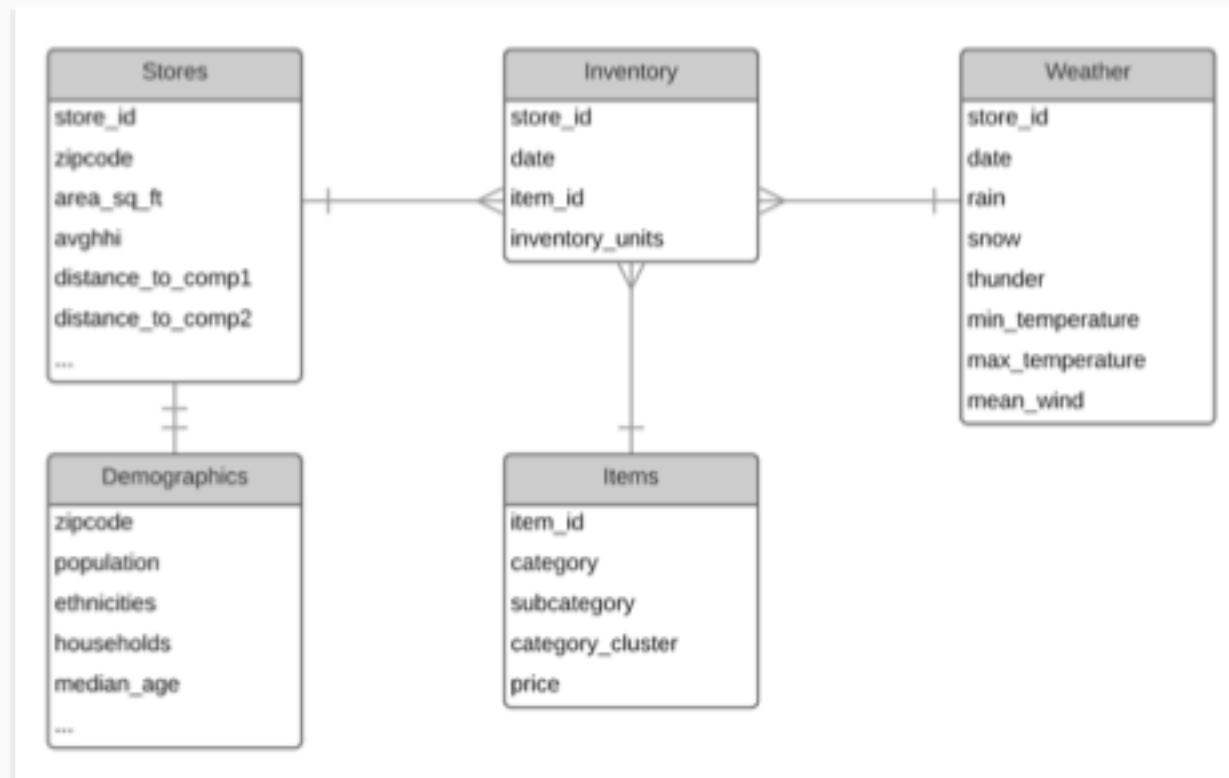
Model	# aggregates
K-means	$\Theta(kn)$
PCA	$\Theta(kn^2)$

k : # clusters

We Efficiently Compute Those Aggregates



Case Study: Retail dataset



Case Study: Retail dataset

Relation	Cardinality (# Tuples)	Degree (# k/v columns)	File size (csv)
Inventory	84,055,817	3 & 1	2 GB
Items	5,618	1 & 4	129 KB
Stores	1,317	1 & 14	139 KB
Demographics	1,302	1 & 15	161 KB
Weather	1,159,457	2 & 6	33 MB
Total:			2.1 GB

Case Study: Retail dataset – PostgreSQL & TensorFlow

- The design matrix is constructed by joining together all the relations
- Train a linear regression model to predict sales by item, store, date from all the other features

Cardinality (# of tuples)	84,055,817
Degree (# of columns)	44 (3 & 41)
Size	23 GB
Time to compute in PostgreSQL	217 secs
Time to export from PostgreSQL	373 secs
Time to learn parameters with GD	> 12,000 secs

Case Study: Retail dataset - comparison

	Design matrix with PostgreSQL/TensorFlow		relationalAI	
	Time	Size	Time	Size
Original	--	2.1 GB	--	2.1 GB
Join Tables	217 secs	23 GB	--	--
Export DM	373 secs	23 GB	--	--
Aggregate	--	--	18 secs	37 KB
Parameter learning with GD	> 12 K secs	--	0.5 secs	--
Total	> 12.5 K secs		18.5 secs	
Improvement (1st Model)	> 676x faster		11x smaller	
Every model after	> 24,000x faster			

Does it work for all model classes or methods?

Supported methods include

- Linear regression
- Polynomial regression
- Factorization machines
- Decision trees
- Linear SVM
- Deep sum-product networks
- Naive Bayes Classifier (discrete case)
- Hidden Markov Model (discrete case)
- K-Means & K-Median clustering
- Gaussian Discriminant Analysis
- Linear Discriminant Analysis
- Principal component analysis
- Frequent item set mining (with Apriori algorithm)
- Computing empirical mutual information and entropy

(with more on the way)

So what?

Some context:



Moore's Law
gives us 2x speedup
every 1.5 years

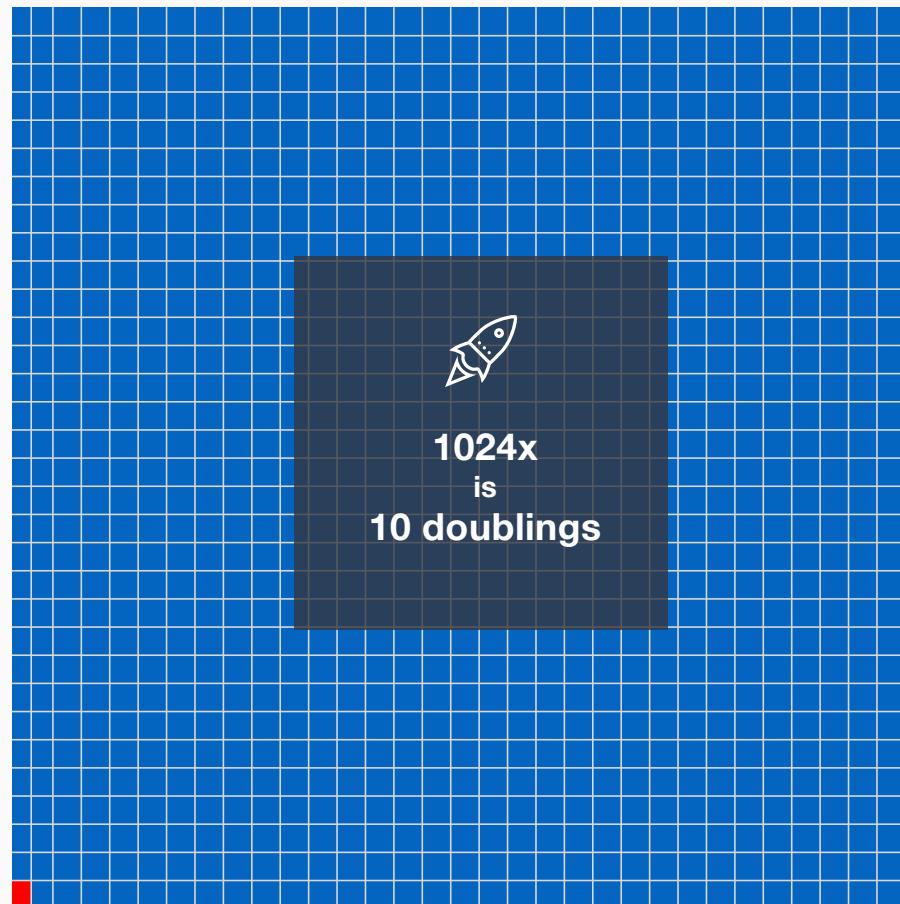
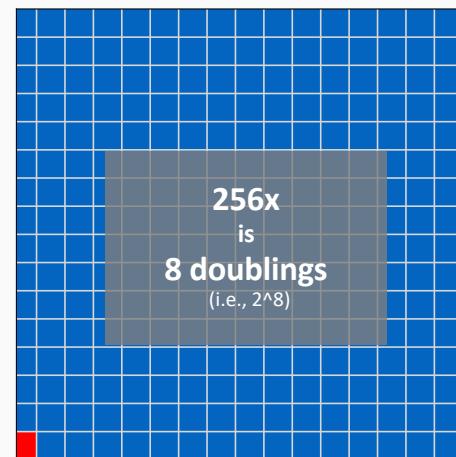


According to Nvidia
GPUs give us a 2-10X
speed-up over CPUs

In other words, GPUs give us ~5 year advantage

So what?

What are the implications of
2-3 orders of magnitude speed-up?



So what?

What are the implications of
2-3 orders of magnitude speed-up?

Algorithms that exploit the domain structure
give us a **12-15 YEAR ADVANTAGE**

256x
is
8 doublings
(i.e., 2^8)

1024x
is
10 doublings
(i.e., 2^{10})

AI's biggest challenges are computational!

ACCURACY

Search for better

- Parameters
- Hyper parameters
- Features
- Models

Don't make assumptions that you don't need to make (e.g. i.i.d. assumption)

VERSATILITY

Reasoning and (generalized) inference: From observations to unknowns in any time period

Inference of any property in the model (e.g., it's just as easy to infer price from sales as it is to infer sales from price)

ROBUSTNESS

Many "big data" problems are really a big collection of small data problems

Overcome challenges with small, incomplete, and dirty data problems by incorporating prior knowledge and expertise

SELF-SUPERVISION

"The future will be self-supervised" Yann LeCun

Build models of the world by observing it and searching model space for the models that have the most explanatory power

INTERPRETABILITY

Searching for models that are accurate and interpretable is harder than searching for accurate models

Interpretation in terms of prior knowledge and in language & ontology that humans understand

EXPLAINABILITY

Explainability typically implanted via separate shadow models that have to be learned

Explanation in terms of prior knowledge and in language & ontology that humans understand

FAIRNESS

It's not enough to exclude gender, ethnicity, race, age, etc as features to the models. Other features might be correlated.

Prejudice is a computational limitation: Reasoning about each person vs reasoning about the group

CAUSALITY

Understanding causality beyond A/B testing

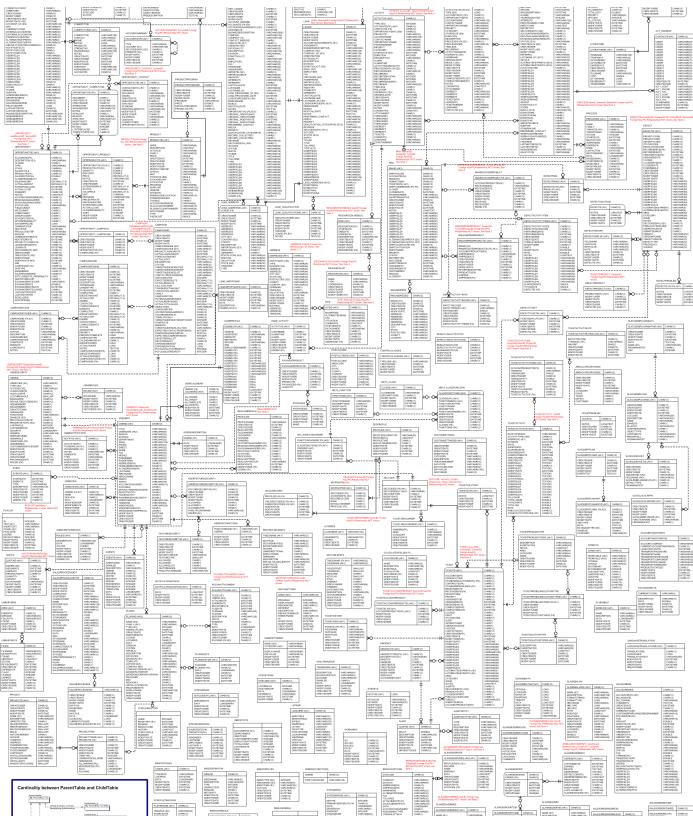
Computationally very expensive

relationalAI

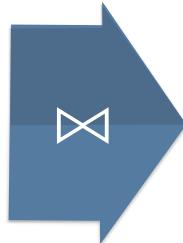
Statistical Relational Learning

Relational generative models

What else **do we throw away** when we build the feature matrix?



Feature extraction query



Translation to feature matrix assumes each entity is independent of the others (iid assumption)

This is often not true - e.g.
related sku's or related people

What if we **don't make the i.i.d assumption?**

Features												
ID	x1	x2	x3	...	y	ID	x1	x2	x3	...	y	

What if we **don't** make the i.i.d assumption?

All

Features

ID	x1	x2	x3	...	y	ID	x1	x2	x3	...	y

...

ID	x1	x2	x3	...	y

Statistical Relational Learning

- Statistical Relational models generalize PGMs in the same way that first order logic generalizes propositional logic
 - they allow us to quantify over individuals/entities
 - Allows for generalization (e.g. item, sub-class, class, dept, etc.)
 - Ability to predict link-based patterns (e.g. inter item dependencies at sub-class, class, dept etc.)
 - Models a varied number of observations for each object/relation. (e.g. friends, colleagues, etc.)
- Variants
 - MLN in various flavors, PSL, RDN, BoostSRL, ProbLog, etc.

Statistical Relational Learning

■ Inference

- Unlike “traditional” methods where prediction is the input applied to the parameters of the model class, inference in SRL requires expensive optimization or (approximate) integration over possible worlds

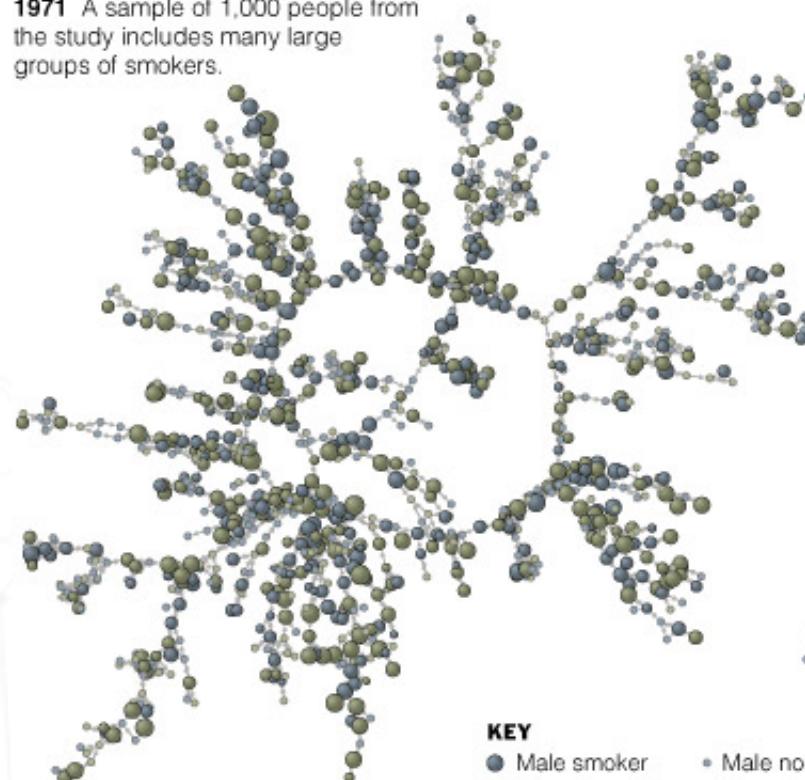
■ Learning

- Unlike traditional learning algorithms, just one instance to learn from (the relational DB)
- Structure learning uses inference during each step

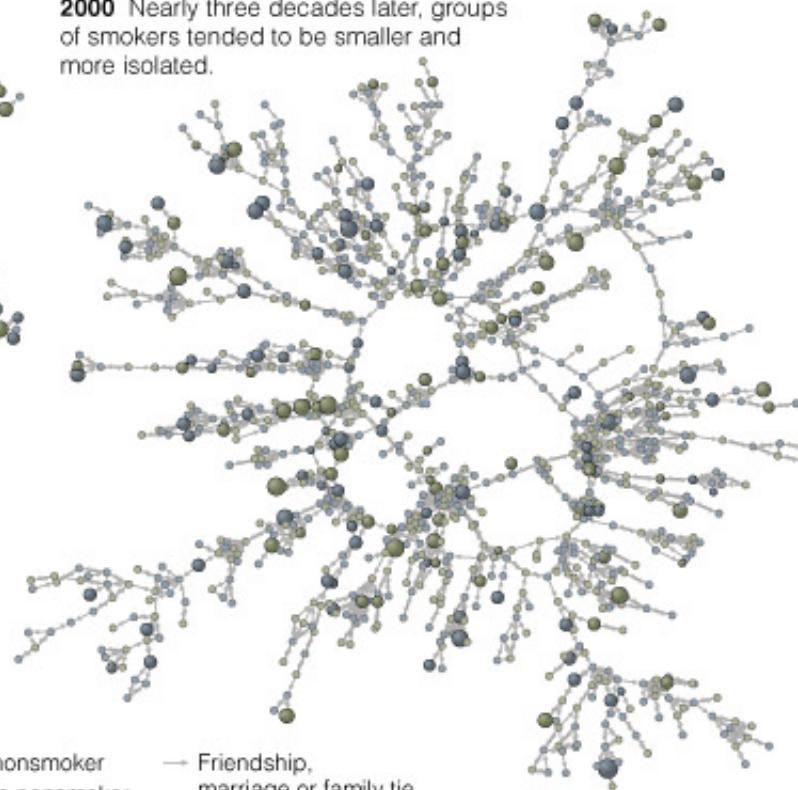
Smoking and Quitting in Groups

Researchers studying a network of 12,067 people found that smokers and nonsmokers tended to cluster in groups of close friends and family members. As more people quit over the decades, remaining groups of smokers were increasingly pushed to the periphery of the social network.

1971 A sample of 1,000 people from the study includes many large groups of smokers.



2000 Nearly three decades later, groups of smokers tended to be smaller and more isolated.



KEY

● Male smoker

● Female smoker

● Male nonsmoker

● Female nonsmoker

→ Friendship,
marriage or family tie

Sources: *New England Journal of Medicine*; Dr. Nicholas A. Christakis; James H. Fowler

Circle size is proportional to the number of cigarettes smoked per day.

THE NEW YORK TIMES

CERTAIN KNOWLEDGE WITH INTEGRITY CONSTRAINTS

A logical Knowledge Base is a set of Integrity Constraints that define a set of possible worlds:

```
person (x)
smokes (x) -> person (x)
cancer (x) -> person (x)
friends (x, y) -> person (x) , person (y)
```

CERTAIN KNOWLEDGE WITH INTEGRITY CONSTRAINTS

A logical Knowledge Base is a set of Integrity Constraints that define a set of possible worlds:

```
person (x)  
smokes (x) -> person (x)  
cancer (x) -> person (x)  
friends (x, y) -> person (x), person (y)
```

Smoking causes cancer
Friends have similar smoking habits

CERTAIN KNOWLEDGE WITH INTEGRITY CONSTRAINTS

A logical Knowledge Base is a set of Integrity Constraints that define a set of possible worlds:

```
person (x)
smokes (x) -> person (x)
cancer (x) -> person (x)
friends (x, y) -> person (x), person (y)
```

Smoking causes cancer
Friends have similar smoking habits

```
w1   smokes (x) -> cancer (x)
w2   smokes (x), friends (x, y) -> smokes (y)
```

How do you make this tractable?

Approximate answer by converting into convex continuous optimization problem

Exploit group symmetry → lifted inference and approximate lifted inference

Avoid grounding altogether → in-database learning

Leveraging database semantics to avoid having to cluster -> in-database SPNs

Stay tuned

relationalAI

Brawn

Do same amount of work faster

The Path to Performance: **Brawn**

Constant factors – Do same amount of work faster (i.e., brawn)

- **Latency hiding:** Memory hierarchy and network latencies (e.g., in memory and near-data computing)
- **Parallelization:** SIMD, multi-core, accelerators (e.g., GPU, TPU, FPGA)
- **Specialization:** Specialize for workload (e.g., JIT compilation), specialize for data

Motivation for implementation strategy

and



3 to 5 years building something similar in prior lives using C++ without ability to specialize for queries or data sets

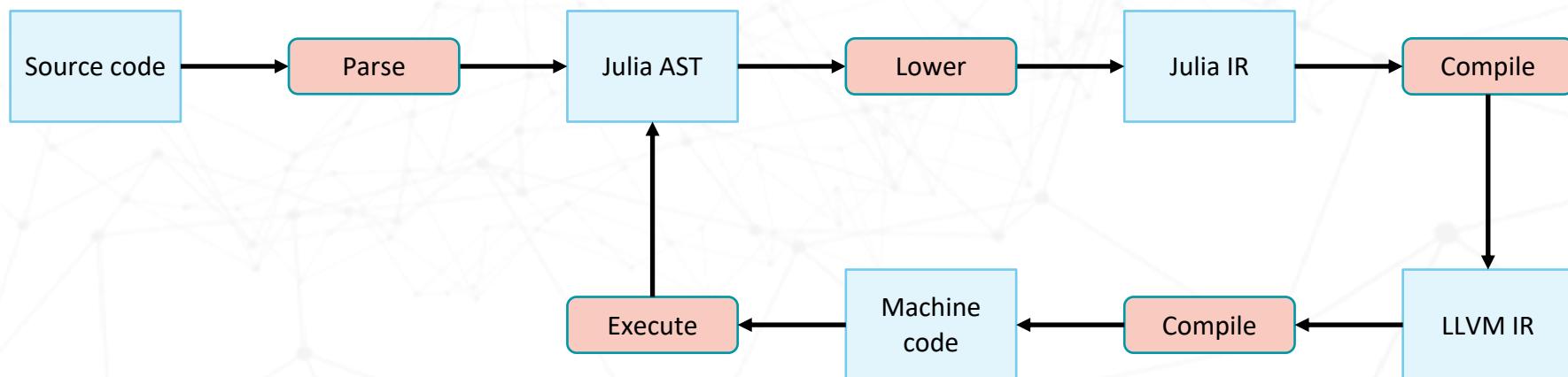


Julia in a nutshell

“Looks like Python, feels like LISP, runs like C”

Julia is fast, dynamic, optionally typed, and multi-dispatched

- Feels like Lisp: Hygienic macros, code quoting, generated functions
- Runs like C: Specialization based on type inference, inlining, unboxing, LLVM to gen assembly



Brains and Brawn: Systems Programming in Julia

- ****Specialization****
 - **Query evaluation:** Just-in-time compiled query plans
- Specialization
 - **Data types:** e.g., fixed-precision decimals

Just-in-Time Query Compilation

- Query compilation has only recently replaced interpretation in modern database systems

```
select A, B, C  
from R, S, T  
where ...  
group by ...
```



```
pushq    %rbp  
movq    %rsp, %rbp  
testq   %rdi, %rdi  
negq    %rdi  
movq    %rdi, %rax  
...  
...
```

- But, state of the practice is surprisingly primitive
 - Typically: variations on template expansion in C/C++
 - Ad-hoc methods to generate code: e.g., write a text file and invoke gcc
 - Cumbersome engineering effort
- Better: use a language with proper staged metaprogramming support
 - e.g., LegoDB using Scala/LMS/Squid
- Julia is very appealing from this point of view!

Simplified TPC-H Q1: from SQL to Julia to Native Code

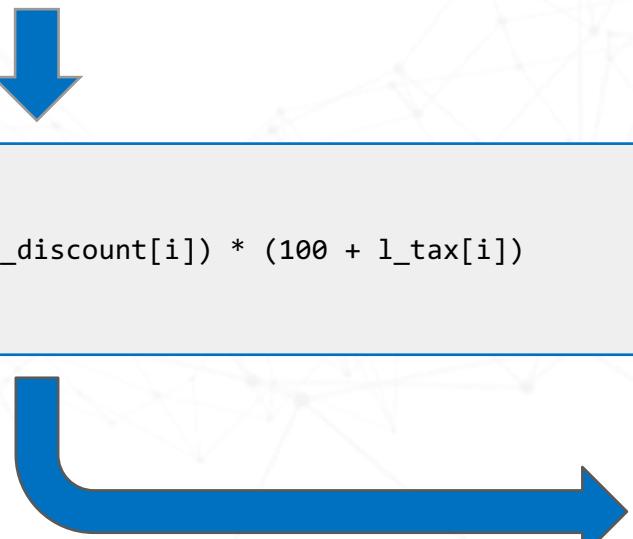
```
select
    sum(l_extprice * (100 - l_discount) * (100 + l_tax))
from
    lineitem
```

From SQL to Julia with
runtime code generation

```
sum = 0
for i in 1:size
    sum += l_extprice[i] * (100 - l_discount[i]) * (100 + l_tax[i])
end
return sum
```

From Julia to LLVM to
optimized x86-64 *

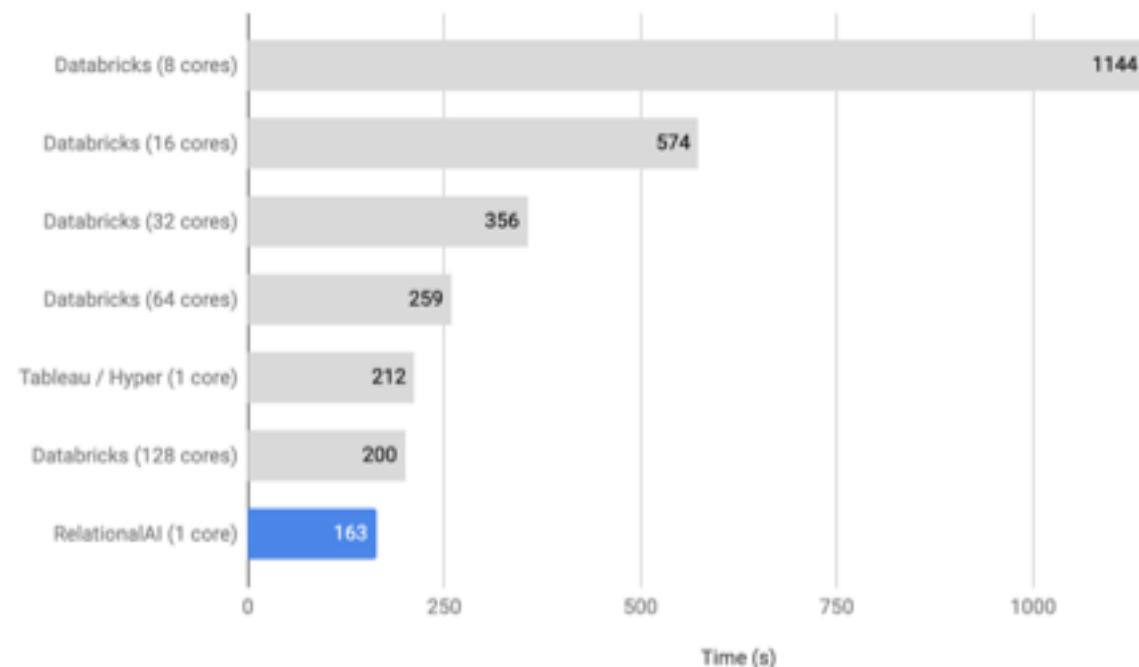
(*) The loop actually even gets vectorized, but we produced simpler
code here for presentation purposes



```
testq %rcx, %rcx
jle L71
movq (%rdi), %r8
movq (%rsi), %r9
movq (%rdx), %r10
xorl %edi, %edi
xorl %eax, %eax
L32:
    movl $100, %esi
    subq (%r9,%rdi,8), %rsi
    movq (%r10,%rdi,8), %rdx
    addq $100, %rdx
    imulq (%r8,%rdi,8), %rsi
    imulq %rdx, %rsi
    addq %rsi, %rax
    addq $1, %rdi
    cmpq %rdi, %rcx
    jne L32
    xorl %eax, %eax
    retq
L71:
    xorl %eax, %eax
    retq
```

BI benchmark: vs Tableau/Hyper and Databricks Spark

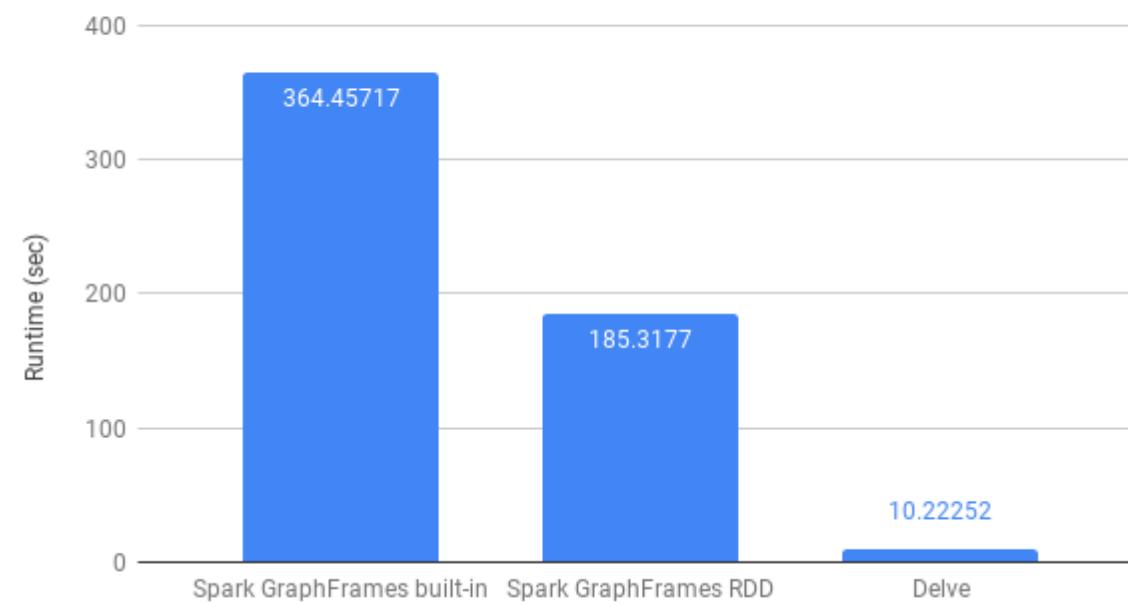
TPC-H Scale Factor 100



Spark numbers based on Databricks hardware and TPCH setup. Snowflake benchmarks closer to Spark than Hyper.

Brains and Brawn Together: 3-Clique Graph benchmark vs Databricks Spark

Triangle Count on graph500 dataset



All benchmarks run on 1 core laptop.

Brains and Brawn: Systems Programming in Julia

- Specialization
 - **Query evaluation:** Just-in-time compiled query plans
- ****Specialization****
 - **Data types:** e.g., fixed-precision decimals

Abstraction without regret by example: Fixed-precision decimals

Fixed-precision decimals are an important data type in database systems (e.g., for currencies), and avoid the inexact representation problems of floats:

```
julia> 0.3333 + 0.33333  
0.6666300000000001 # oops
```

The Julia ecosystem has a FixedPointDecimal package for this purpose

```
julia> T = FixedDecimal{Int64,5}  
FixedDecimal{Int64,5}  
  
julia> T(0.3333) + T(0.33333)  
FixedDecimal{Int64,5}(0.66663) # much better!
```

But... is this really going to be efficient enough? (Most database systems need special code to “compile away” fixed precision decimal operations into simple operations on integers...)

Here's the FixedDecimal datatype and its addition operation...

```
struct FixedDecimal{T <: Integer, f} <: Real
    i::T

    function Base.reinterpret(::Type{FixedDecimal{T, f}}, i::Integer) where {T, f}
        n = max_exp10(T)
        if f >= 0 && (n < 0 || f <= n)
            new{T, f}(i % T)
        else
            _throw_storage_error(f, T, n)
        end
    end
end

+(x::FixedDecimal{T, f}, y::FixedDecimal{T, f}) where {T, f} =
    reinterpret(FD{T, f}, x.i+y.i)
```

... and lo, the Julia compiler produces a tiny # of ops on integers, just as required!

```
julia> @code_native +(T(0.3333), T(0.33333))
decl %eax
movl (%esi), %eax
decl %eax
addl (%edi), %eax
retl
```

Moreover, this will be inlined
at the call site in any practical
example!

■ What about Parallelization and Accelerators?

» Manual » Parallel Computing

[Edit on GitHub](#)

Parallel Computing

For newcomers to multi-threading and parallel computing it can be useful to first appreciate the different levels of parallelism offered by Julia. We can divide them in three main categories :

1. Julia Coroutines (Green Threading)
2. Multi-Threading
3. Multi-Core or Distributed Processing

We will first consider Julia [Tasks \(aka Coroutines\)](#) and other modules that rely on the Julia runtime library, that allow us to suspend and resume computations with full control of inter-Tasks communication without having to manually interface with the operating system's scheduler. Julia also supports communication between Tasks through operations like `wait` and `fetch`. Communication and data synchronization is managed through [Channels](#), which are the conduits that provide inter-Tasks communication.

Julia also supports experimental multi-threading, where execution is forked and an anonymous function is run across all threads. Known as the fork-join approach, parallel threads execute independently, and must ultimately be joined in Julia's main thread to allow serial execution to continue. Multi-threading is supported using the `Base.Threads` module that is still considered experimental, as Julia is not yet fully thread-safe. In particular segfaults seem to occur during I/O operations and task switching. As an up-to-date reference, keep an eye on the [issue tracker](#). Multi-Threading should only be used if you take into consideration global variables, locks and atomics, all of which are explained later.

In the end we will present Julia's approach to distributed and parallel computing. With scientific computing in mind, Julia natively implements interfaces to distribute a process across multiple cores or machines. Also we will mention useful external packages for distributed programming like `MPI.jl` and `DistributedArrays.jl`.

High-level GPU programming in Julia

Tim Besard

Computer Systems Lab
Ghent University, Belgium
Tim.Besard@elis.ugent.be

Pieter Verstraete

Ghent University, Belgium

Bjorn De Sutter

Computer Systems Lab
Ghent University, Belgium
Bjorn.DeSutter@elis.ugent.be

Abstract

GPUs are popular devices for accelerating scientific calculations. However, as GPU code is usually written in low-level languages, it breaks the abstractions of high-level languages popular with scientific programmers. To overcome this, we present a framework for CUDA GPU programming in the high-level Julia programming language. This framework compiles Julia source code for GPU execution, and takes care of the necessary low-level interactions using modern code generation techniques to avoid run-time overhead.

Evaluating the framework and its APIs on a case study comprising the trace transform from the field of image processing, we find that the impact on performance is minimal, while greatly increasing programmer productivity. The metaprogramming capabilities of the Julia language proved invaluable for enabling this. Our framework significantly improves usability of GPUs, making them accessible for a wide range of programmers. It is available as free and open-source software licensed under the MIT License.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Code generation, Compilers, Runtime environments

Keywords Julia, GPU, CUDA, LLVM, Metaprogramming

1. Introduction

GPUs can significantly speed up certain workloads. However, targeting GPUs requires serious effort. Specialized machine code needs to be generated through the use of a vendor-supplied compiler. Because of the architectural set-up, initiating execution on the coprocessor is often quite complex as well. Even though the vendors try hard to supply toolchains that support different developer environments and offer convenience functionality to lower the burden, they are essentially playing catch-up.

While coprocessor hardware improves program efficiency, high-level languages are becoming a popular choice because of their improved programmer productivity. Languages such as Python or Julia provide a user-friendly development environment. Low-level details are hidden from view, and secondary tasks such as dependency management and compiling and linking are automatically taken care of.

For users of these high-level languages, jumping through the many hoops of GPU development is often an exceptionally large burden. A lot of low-level knowledge is required, and many of the user-friendly abstractions break down. For example, when using Python to target NVIDIA GPU's using the CUDA toolkit, the developer needs to write GPU kernels in CUDA C, and interact with the CUDA API in order to compile the code, prepare the hardware and launch the kernel. The situation is even worse for languages unsupported by the CUDA toolkit, such as Julia, in which case there are only superficial or no CUDA API wrappers at all.

Ideally, it should be possible to develop and execute high-level GPU kernels without much extra effort: writing kernels in high-level source code, while the interpreter for that language takes care of compiling the necessary functions to GPU machine code. Low-level details should be automated, or at least wrapped in user-friendly language constructs.

This paper presents a framework to target NVIDIA GPUs, and by extent other accelerators, directly in the Julia programming language. Kernels can be written in high-level Julia code. We also created high-level CUDA API wrappers to support the natural use of the CUDA API from within Julia. The framework provides a user-friendly GPU kernel programming and execution interface that automates driver interactions and abstracts GPU-specific details without introducing any run-time overhead. All code implementing this framework is available as open-source code on GitHub.

In Section 2 we describe relevant technologies and the motivation for our work. Section 3 provides an overview of our framework, each component explained in detail in Sections 4 to 6. Finally, we evaluate our work in Section 7.

[Copyright notice will appear here once 'preprint' option is removed.]

1

2016/4/13

AUTOMATIC FULL COMPIRATION OF JULIA PROGRAMS AND ML MODELS TO CLOUD TPUS

Keno Fischer ¹ Elliot Saba ¹

ABSTRACT

Google's Cloud TPUs are a promising new hardware architecture for machine learning workloads. They have powered many of Google's milestone machine learning achievements in recent years. Google has now made TPUs available for general use on their cloud platform and as of very recently has opened them up further to allow use by non-TensorFlow frontends. We describe a method and implementation for offloading suitable sections of Julia programs to TPUs via this new API and the Google XLA compiler. Our method is able to completely fuse the forward pass of a VGG19 model expressed as a Julia program into a single TPU executable to be offloaded to the device. Our method composes well with existing compiler-based automatic differentiation techniques on Julia code, and we are thus able to automatically obtain the VGG19 backwards pass and similarly offload it to the TPU. Targeting TPUs using our compiler, we are able to evaluate the VGG19 forward pass on a batch of 100 images in 0.23s which compares favorably to the 52.4s required for the original model on the CPU. Our implementation is less than 1000 lines of Julia, with no TPU specific changes made to the core Julia compiler or any other Julia packages.

1 INTRODUCTION

One of the fundamental changes that has enabled the steady progress of machine learning techniques over the past several years has been the availability of vast amounts of compute power to train and optimize machine learning models. Many fundamental techniques are decades old, but only the compute power available in recent years was able to deliver sufficiently good results to be interesting for real world problems. A significant chunk of this compute power has been available on Graphics Processing Units (GPUs) whose vector compute capability, while originally intended for graphics have shown to deliver very good performance on the kind of matrix-heavy operations generally performed in machine learning models.

In this paper, we present initial work to compile general Julia code to TPU using this interface. This approach is in contrast to the approach taken by TensorFlow (Abadi et al., 2016), which does not compile Python code proper, but rather uses Python to build a computational graph, which is then compiled. It is aesthetically similar to JAX (Frostig et al., 2018), which does aim to offload computations written in Python proper by tracing and offloading high-level array operations. Crucially, however, we do not rely on tracing, instead we leverage Julia's static analysis and compilation capabilities to compile the full program, including any control flow to the device. In particular, our approach allows users to take advantage of the full expressiveness of the Julia programming language in writing their models. This includes higher-level features such as multiple dispatch, higher order functions and existing libraries such as those for differential equation solvers (Rackauckas & Nie, 2017) and generic linear algebra routines. Since it operates on pure

accelerator available to the public via their cloud offering. Originally, the use of TPUs was restricted to applications written using Google's TensorFlow machine learning framework. Fortunately, in September 2018, Google opened up access to TPUs via the IR of the lower level XLA ("Accelerated Linear Algebra") compiler. This IR is general purpose and is an optimizing compiler for expressing arbitrary computations of linear algebra primitives and thus provides a good foundation for targeting TPUs by non-Tensorflow users as well as for non-machine learning workloads.

In this paper, we present initial work to compile general Julia code to TPU using this interface. This approach is in contrast to the approach taken by TensorFlow (Abadi et al., 2016), which does not compile Python code proper, but rather uses Python to build a computational graph, which is then compiled. It is aesthetically similar to JAX (Frostig et al., 2018), which does aim to offload computations written in Python proper by tracing and offloading high-level array operations. Crucially, however, we do not rely on tracing, instead we leverage Julia's static analysis and compilation capabilities to compile the full program, including any control flow to the device. In particular, our approach allows users to take advantage of the full expressiveness of the Julia programming language in writing their models. This includes higher-level features such as multiple dispatch, higher order functions and existing libraries such as those for differential equation solvers (Rackauckas & Nie, 2017) and generic linear algebra routines. Since it operates on pure

In 2017, Google announced that they would make their proprietary Tensor Processing Unit (TPU) machine learning

¹Julia Computing, Inc.. Correspondence to: Keno Fischer <keno@juliacomputing.com>.

Preliminary work.

relationalAI

Closing

One more time

AI's biggest opportunities are relational!

ACCURACY

Search for better

- Parameters
- Hyper parameters
- Features
- Models

Don't make assumptions that you don't need to make (e.g. i.i.d. assumption)

VERSATILITY

Reasoning and (generalized) inference: From observations to unknowns in any time period

Inference of any property in the model (e.g., it's just as easy to infer price from sales as it is to infer sales from price)

ROBUSTNESS

Many "big data" problems are really a big collection of small data problems

Overcome challenges with small, incomplete, and dirty data problems by incorporating prior knowledge and expertise

SELF-SUPERVISION

"The future will be self-supervised" Yann LeCun

Build models of the world by observing it and searching model space for the models that have the most explanatory power

INTERPRETABILITY

Searching for models that are accurate and interpretable is harder than searching for accurate models

Interpretation in terms of prior knowledge and in language & ontology that humans understand

EXPLAINABILITY

Explainability typically implanted via separate shadow models that have to be learned

Explanation in terms of prior knowledge and in language & ontology that humans understand

FAIRNESS

It's not enough to exclude gender, ethnicity, race, age, etc as features to the models. Other features might be correlated.

Prejudice is a computational limitation: Reasoning about each person vs reasoning about the group

CAUSALITY

Understanding causality beyond A/B testing

Computationally very expensive

Why hasn't this happened yet?

AI investment is focused on consumer AI

- Deep learning for images, speech, text → not relational data (yet)

Weaknesses of implementations of relational data management systems

- Abstraction leads to regret
- Can guarantee correct answer but can't guarantee optimal path to get there
- Limitations on expressiveness, i.e. I can't always ask the question I want to ask

Inertia — we have something that (sort of) works and we're getting by. “you can't expect us to rewrite all this code and retrain all those data scientists and programmers”

- The number of models that haven't been built is **>>>** the number of models that have
- The number of future modelers is **>>>** the number of current modelers
- The number of domain experts is **>>>** the number of modelers and data scientists

Why Now?

- We invented a new generation of (meta) algorithms that provide optimal solutions to large problem classes
 - OOM **more power** for OOM **better intelligence**
- New generation of compilers that eliminate the cost of abstraction
 - Allow us to specialize for workload
 - Allow us to specialize for datasets
- Backlash against Hadoop (Map-Reduce), NoSQL, ML Frameworks – “the emperor has no clothes” is in the air
 - Require you to sell your soul for scalability and/or performance
 - Harder to program and operate

What are we doing about it?

We built a system that gives you abstraction without regret

How are we going to do that?

- Constant factors
- Asymptotic factors

We're going to meet people where they are:

- Tables and SQL if you are an analyst
- Tensors & Linear Algebra if you are a data scientist

We're going to simplify and consolidate analytics:

- The building blocks for next gen AI (e.g. fast aggregation, factoring, multi-way evaluation, JIT, accelerators) building blocks for all enterprise analytics: BI, graphs, rules, planning, mathematical optimization.

We're going to stage it. We're going to consolidate and checkpoint our gains as we go.

- AutoML (with automatic feature engineering and relational statistics) -> Data scientist
- Data Management Systems for Analytics (aka data lakes) -> Data scientist
- Business Intelligence & Data Warehouses -> Analyst & End User

Product: Never have to start from scratch again

	<h3>Data</h3> <ul style="list-style-type: none">General: e.g. Weather, Events, Consumer, SentimentDomain and industry specific: e.g. securities, crypto currenciesCompetitor: e.g. price
	<h3>Templates</h3> <ul style="list-style-type: none">Industry: retail, financial services, technology & software.Problem class: (product) knowledge graphs, recommender systems, anomaly detection, portfolio optimization
	<h3>Tools</h3> <ul style="list-style-type: none">Data scientists: Notebooks (e.g. Jupyter)Domain modelers: e.g. ontology editors (e.g. Jupyter, NORMA, Protégé)Analysts: e.g. BI and spreadsheets
	<h3>Engine</h3> <ul style="list-style-type: none">DatabaseAI and Analytics

References

Incomplete list

Underlying magic: Worst-case optimal join algorithms

- Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. Ngo. (Gems of PODS 2018)
- Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. Ngo, Porat, Re, Rudra. (Journal of the ACM 2018)
- What do Shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? Abo Khamis, Ngo, Suciu, (PODS 2017 - Invited to Journal of ACM)
- Computing Join Queries with Functional Dependencies. Abo Khamis, Ngo, Suciu. (PODS 2017)
- Joins via Geometric Resolutions: Worst-case and Beyond. Abo Khamis, Ngo, Re, Rudra. (PODS 2015, Invited to TODS 2015)
- Beyond Worst-Case Analysis for Joins with Minesweeper. Abo Khamis, Ngo, Re, Rudra. (PODS 2014)
- Leapfrog Triejoin: A Simple Worst-Case Optimal Join Algorithm. Veldhuizen (ICDT 2014 - Best Newcomer)
- Skew Strikes Back: New Developments in the Theory of Join Algorithms. Ngo, Re, Rudra. (Invited to SIGMOD Record 2013)
- Worst Case Optimal Join Algorithms. Ngo, Porat, Re, Rudra. (PODS 2012 – Best Paper)



Underlying magic: Optimal query plans for worst-case optimal joins

- Juggling functions inside a database, Abo Khamis, Ngo, Suciu (Invited to SIGMOD Record)
 - On Functional Aggregate Queries with Additive Inequalities. Abo Khamis, Curtin, Moseley, Ngo, Nguyen, Olteanu, Schleich. PODS 2019
 - What do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog have to do with one another? Abo Khamis, Ngo, Suciu, (PODS 2017 - Invited to Journal of ACM)
 - FAQ: Questions Asked Frequently, Abu Khamis, Ngo, Rudra, (PODS 2016 – Best Paper, Invited to Journal of ACM)

Underlying magic: In-database relational learning

- Rk-means: Fast Clustering for Relational Data. Curtin, Moseley, Ngo, Nguyen, Olteanu, Schleich. Submitted to NeurIPS 2019
- On coresets for logistic regression. Curtin, Moseley, Pruhs, Samadian. Submitted to NeurIPS 2019
- SolverBlox: Algebraic Modeling in Datalog. Borraz-Sanchez, Klabjan, Pasalic, Aref. (Declarative Logic Programming – Morgan & Claypool 2018)
- In-Database Learning with Sparse Tensors, Abo Khamis, Ngo, Nguyen, Olteanu, Schleich (PODS 2018 - Invited to Journal of TODS)
- AC/DC: In-Database Learning Thunderstruck, Abo Khamis, Ngo, Nguyen, Olteanu, Schleich (DEEM 2018)
- Modelling Machine Learning Algorithms on Relational Data with Datalog. Makrynioti, Vasiloglou, Pasalic, Vassalos. (DEEM 2018)
- In-Database Factorized Learning, Ngo, Nguyen, Olteanu, Schleich (AMW 2017)
- Data Science with Linear Programming. Makrynioti, Vasiloglou, Pasalic, Vassalos. (DeLBP 2017)

In-Database Factorized Learning

Hang Q. Ngo¹, Tran Cong Nguyen², Eric Ohnsorge³, and Matthias Aebischer²

In-Database Learning with Sparse Tensors

Matthias Aebischer¹, Hang Q. Ngo², and Tran Cong Nguyen²

AC/DC: In-Database Learning Thunderstruck

Matthias Aebischer¹, Hang Q. Ngo², and Tran Cong Nguyen²

Abstract

In this paper, we propose a novel framework for learning with relational data. We introduce a factorized learning model that can learn from relational data in an in-database setting. This model is based on sparse tensors, which are tensors with many zero entries. Sparse tensors provide a natural way to represent relational data, such as graphs and networks. We propose a new algorithm for learning with sparse tensors, called AC/DC, which is able to learn from relational data in an in-database setting. The AC/DC algorithm is able to learn from large datasets and is able to handle complex relational data. The AC/DC algorithm is able to learn from relational data in an in-database setting, which is a challenging task. The AC/DC algorithm is able to learn from relational data in an in-database setting, which is a challenging task. The AC/DC algorithm is able to learn from relational data in an in-database setting, which is a challenging task.

Abstract

We propose a novel framework for learning with relational data. We introduce a factorized learning model that can learn from relational data in an in-database setting. This model is based on sparse tensors, which are tensors with many zero entries. Sparse tensors provide a natural way to represent relational data, such as graphs and networks. We propose a new algorithm for learning with sparse tensors, called AC/DC, which is able to learn from relational data in an in-database setting. The AC/DC algorithm is able to learn from large datasets and is able to handle complex relational data. The AC/DC algorithm is able to learn from relational data in an in-database setting, which is a challenging task. The AC/DC algorithm is able to learn from relational data in an in-database setting, which is a challenging task.

Abstract

We propose a novel framework for learning with relational data. We introduce a factorized learning model that can learn from relational data in an in-database setting. This model is based on sparse tensors, which are tensors with many zero entries. Sparse tensors provide a natural way to represent relational data, such as graphs and networks. We propose a new algorithm for learning with sparse tensors, called AC/DC, which is able to learn from relational data in an in-database setting. The AC/DC algorithm is able to learn from large datasets and is able to handle complex relational data. The AC/DC algorithm is able to learn from relational data in an in-database setting, which is a challenging task. The AC/DC algorithm is able to learn from relational data in an in-database setting, which is a challenging task.

Underlying magic: Julia

- Julia: Dynamism and Performance Reconciled by Design, Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Lionel Zoubritzky, Jan Vitek (OOPSLA 2018)
 - Julia Subtyping: A Rational Reconstruction, Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, Jan Vitek (OOPSLA 2018)
 - Julia: A fresh approach to numerical computing, Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah (SIAM Review 2017)

SAM REVIEW
Vol. 59, No. 1, pp. 65–98

© 2017 Society for Industrial and Applied Mathematics

Julia: A Fresh Approach to Numerical Computing*

Julia: Dynamism and Performance Reconciled by Design

JEFF BEZANSON, Julia Comput

JIAHAO CHEN, Capital One

BEN CHUNG, Northeastern Uni

STEFAN KARPINSKI, Julia Con

VIRAL B. SHAH, Julia Computin

LIONEL ZOBRITZKY, École

JAN VITEK, Northeastern Univer

Abstract. Bridging cultures that fields of computer science computing. Julia is de “laws of nature” by

1. High-level dynamic
2. One must prototype or deployment.
3. There are parts of best left untouched.

We introduce the Juliaization and abstraction a technique from comp. Abstraction, which is w same after differences code through another t Julia shows that one convenience.

Key words. Julia, numerical, scie

AMS subject classifications. 68N1

DOI. 10.1137/141000671

Contents

I Scientific Computing L

1.1 Julia Architecture a

*Received by the editors Dec 16, 2015; published electronically http://www.siam.org/journals/

Funding: This work received f Innovation, the Intel Science and Singapore MIT Alliance, an Anu DMS-1016125, and DMS-131283 Columbia University for petascale Dogru and Shell Oil thanks to Al Analysis, Chris Mertzl, and the

[†]Julia Computing, Inc. (jeff@csail.mit.edu)

[‡]CSAIL and Department of M 02139 (edelman@math.mit.edu).

[§]New York University, New Yo com).

Julia is a programming language for t such as Python or MATLAB, with cl Julia’s productivity features include: and multiple dispatch. At the same specializing just-in-time compiler to design choices made by the creators and usability.

CCS Concepts: • Software and its just-in-time compilers; Multiparadig

Additional Key Words and Phrases:

ACM Reference Format:

Jeff Bezanson, Jiahaoy Chen, Ben Chu Julia: Dynamism and Performance | (2018), 23 pages. <https://doi.org/10.1137/141000671>

1 INTRODUCTION

Scientific programming has traditionally been done in high-level productivity languages (Python, C++, Fortran) for speed and a precise type system. Julia is designed as dynamic typing or garbage collection, as well as dynamic complexity often turn to performance languages. existing application (or some sub-involvement; features previously to be emulated by hand. As a result, this can be a daunting task.

Scientists have been trying to One example is the ROOT data petabytes of data, the high energy extension to C++, providing inte

Permission to make digital or hard copies provided that copies are not made or distributed outside the first page. Copyrig contact the owner/authors.

© 2018 Copyright held by the owner/auth

https://doi.org/10.1137/141000671

https://doi.org/0.0000/00000000

Proceedings of the ACM on

Julia Subtyping: A Rational Reconstruction

FRANCESCO ZAPPA NARDELLI, Inria and Northeastern U.

JULIA BELYAKOVA, Czech Technical U. in Prague

ARTEM PELENITSYN, Czech Technical U. in Prague

BENJAMIN CHUNG, Northeastern U.

JEFF BEZANSON, Julia Computing

JAN VITEK, Northeastern U. and Czech Technical U. in Prague

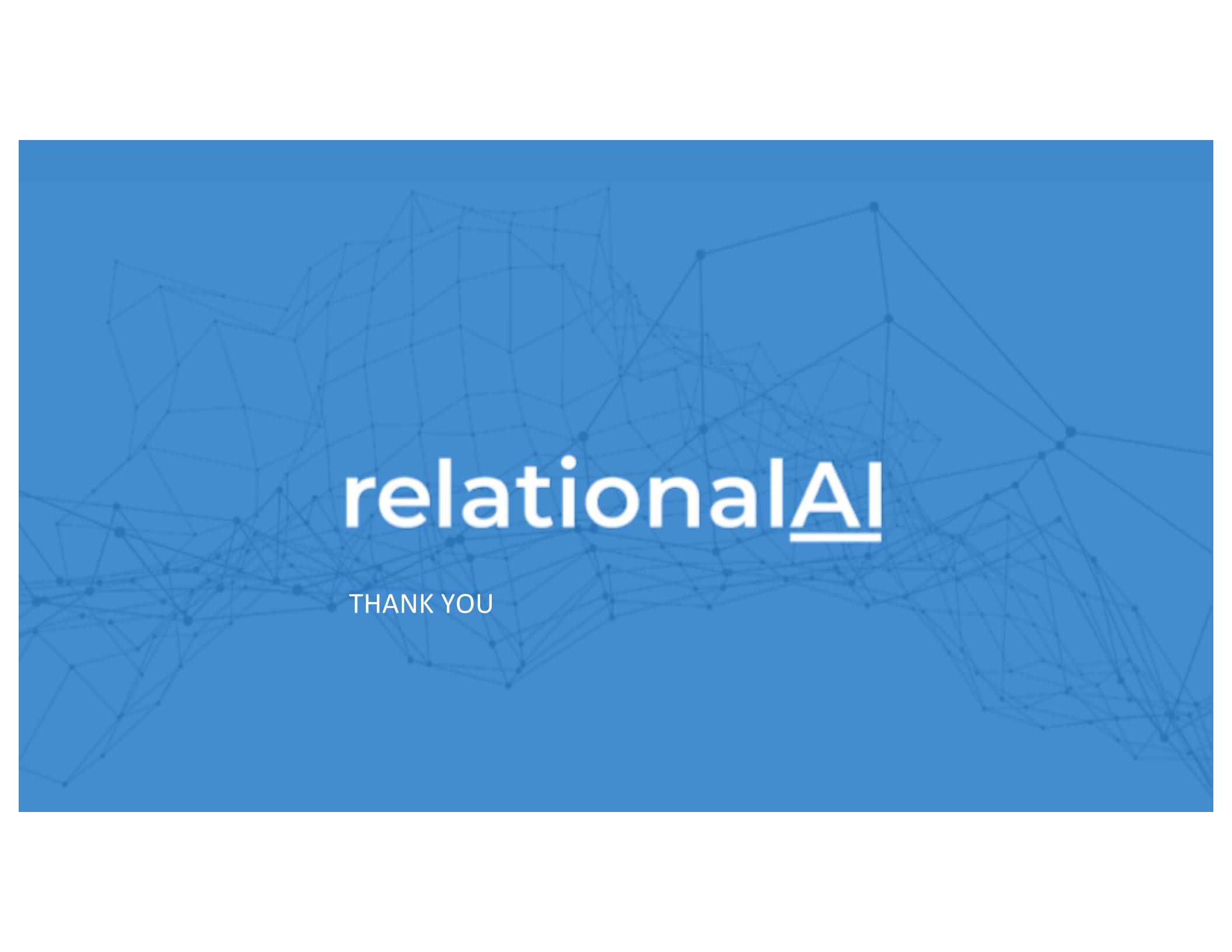
Programming languages that support multiple dispatch rely on an expressive notion of subtyping to specify method applicability. In these languages, type annotations on method declarations are used to select, out of a potentially large set of methods, the one that is most appropriate for a particular tuple of arguments. Julia is a language for scientific computing built around multiple dispatch and an expressive subtyping relation. This paper provides the first formal definition of Julia’s subtype relation and motivates its design. We validate our specification empirically with an implementation of our definition that we compare against the existing Julia implementation on a collection of real-world programs. Our subtype implementation differs on 122 subtype tests out of 6,014,476. The first 120 differences are due to a bug in Julia that was fixed once reported; the remaining 2 are under discussion.

1 INTRODUCTION

Multiple dispatch is used in languages such as CLOS [DeMichiel and Gabriel 1987], Perl [Randall et al. 2003], R [Chambers 2014], Fortress [Allen et al. 2011], and Julia [Bezanson 2015]. It allows programmers to overload a generic function with multiple methods that implement the function for different type signatures; invocation of the function is resolved at run-time depending on the expressive types of the arguments. The expressive power of multiple dispatch stems from the way it constrains the applicability of a method to a particular set of values. With it, programmers can write code that is concise and clear, as special cases, such as optimized versions of matrix multiplication, can be relegated to dedicated methods. The inset shows three of the 181 methods implementing multiplication in Julia’s standard library. The first method implements the case where a range is multiplied by a number. The second method is invoked on generic numbers: it explicitly converts the arguments to a common type via the promote function. The last method invokes native multiplication; its signature has a type variable T that can be instantiated to any integer type.

For programmers, understanding multiple dispatch requires reasoning about the subtype relation. Consider the infix call $x \cdot y$. If x is bound to a float, only the second method is applicable. If instead, x is an integer, then two methods are applicable and Julia’s runtime must identify the *most specific* one. Now, consider $3 \cdot 4$, with argument type `Tuple(Int, Int)`. The signature of the first method is `Tuple(Number, Range)`. Tuples are covariant, so the runtime checks that `Int <: Number` and `Int <: Range`. Integers are subtypes of numbers, but not of ranges, so the first method is not applicable, but the second is, as `Tuple(Int, Int) <: Tuple(Number, Number)`. The third method is also applicable, as `Tuple(Int, Int)` is a subtype of `Tuple(T, T)` where `T <: Union(Signed, Unsigned)`; because there exists an instance of the variable T (namely `Int`) for which subtyping holds. As multiple methods are applicable, subtyping is used to compare their signatures; it holds that `Tuple(T, T)` where `T <: Union(Signed, Unsigned)` is a subtype of `Tuple(Number, Number)` because this holds for all instances of the variable T. The call will be dispatched, as expected, to the third method.

Proceedings of the ACM on Programming Languages, Vol. 1, No. CONF, Article 1. Publication date: January 2018.

A complex, abstract network graph is visible in the background, composed of numerous small, semi-transparent blue dots connected by thin, light blue lines. The graph forms a dense, organic shape that tapers towards the bottom right corner.

relationalAI

THANK YOU