Towards Cost-Optimal Query Processing in the Cloud

Viktor Leis
viktor.leis@fau.de
Friedrich-Alexander-Universität Erlangen-Nürnberg

Maximilian Kuschewski
maximilian.kuschewski@fau.de
Friedrich-Alexander-Universität Erlangen-Nürnberg

ABSTRACT

Public cloud providers offer hundreds of heterogeneous hardware instances. For analytical query processing systems, this presents a major challenge: depending on the hardware configuration, performance and cost may differ by orders of magnitude. We propose a simple and intuitive model that takes the workload, hardware, and cost into account to determine the optimal instance configuration. We discuss how such a model-based approach can significantly reduce costs and also guide the evolution of cloud-native database systems to achieve our vision of *cost-optimal query processing*.

PVLDB Reference Format:
Viktor Leis and Maximilian Kuschewski. Towards Cost-Optimal Query Processing in the Cloud. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Much of the work on query processing minimizes runtime on a particular hardware configuration. Examples are research on distributed query processing using Infiniband [7] or in-memory query processing for NUMA machines [15]. The implicit underlying assumption is that the hardware configuration is more or less fixed, and that the objective is to optimize the software design for that hardware. This perspective comes from a traditional procurement process where servers are acquired under a multi-year time horizon and can therefore be thought of as fixed.

While the assumption of fixed hardware may be reasonable in the on-premise world, in the cloud the situation is different. Per-second billing makes moving applications between instances possible, and public cloud providers offer hundreds of heterogeneous instance types. The quantitative differences between instance types are large and they differ across multiple dimensions as well as price. For example, looking at CPU cores per dollar in AWS EC2, the best option (c5) is 4.9 times cheaper than the most expensive one (x1e). For DRAM capacity the factor is 3.5 (x1e vs. c5d) and for network bandwidth 28.5 (c5n vs. x1e). This makes selecting an instance difficult: depending on the workload characteristics (e.g., CPU-bound or network-bound) any of these instances may be best.

Consider distributed data warehouse systems like Redshift [11] or Snowflake [9] running large analytical queries in the cloud:

- Would it be better to use a balanced configuration or specifically focus on one resource (CPU, network, SSD, or DRAM)?
- Is a single large instance preferable to several smaller ones?
- Which public cloud (AWS, Azure, GCP) is cheapest?

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.

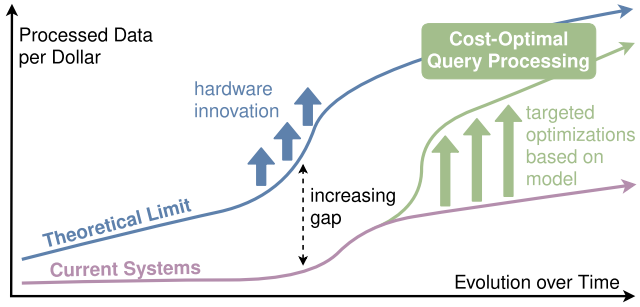


Figure 1: Model-based hardware/software co-evolution

- Should one cache data on instances or always read it on demand from a distributed file system?
- Would it be better to invest engineering effort into optimizing networking or the storage I/O stack?

To answer such questions rigorously, one needs to take an economic perspective. In the cloud, resources (e.g., CPU cores, DRAM, SSD, network) are fungible and can be traded against each other, with money serving as the medium of exchange. Minimizing runtime is not per se the optimization goal, but rather workload cost in dollars given some runtime constraints. Focusing on workload cost directly leads to the notion of *cost optimality*, which is the minimum monetary cost for executing a particular workload in some cloud hardware landscape.

Current cloud systems are still far away from cost optimality. To provide evidence for this claim, let us compare the cost of a hypothetical query engine running on an AWS EC2 instance with AWS Athena. Our example query scans 1 TB of data from S3. If we execute this query on c5n.18xlarge, we have 72 vCPUs and a peak network bandwidth of 100 Gbit. At a realistic rate of 80 Gbit/s, scanning 1 TB of data from S3 takes 100 seconds. During these 100 seconds, the query engine has to process 10 GB/s using the 72 vCPUs, amounting to 142 MB/s per vCPU. Modern query engines can process data at this rate even for moderately complex queries. This means that the execution of our 1 TB query is likely network-bound and takes 100 seconds. If, exploiting per-second billing, we shut down the instance after 100 seconds, the hourly on demand c5n.18xlarge price of $3.89 results in a total query cost of only $0.11. The cost of Athena, on the other hand, is $5 for the same query, i.e., in this scenario Athena is almost 50× more expensive. The use of *spot* or *reserved* rather than *on demand* instances would increase the gap further to 100-200×.

Our long-term vision is to build a cloud-native OLAP system that approaches cost optimality, as shown in Figure 1. We believe that in order to achieve this ambitious goal we need a model that estimates how much a particular workload would cost on some specific hardware configuration (similar to the implicit model in the previous paragraph). The key feature of such a model is that it

assumes a hypothetical system that is capable of fully exploiting all available hardware resources, rather than modeling any specific, existing database system with all its performance bottlenecks. In other words, the model provides a lower bound for workload cost.

The model is meant as a stepping stone for building a cost-optimal OLAP system in the cloud. Developing any industrial-strength database system is difficult and requires a multi-year effort. In the past decade, one could observe rapid changes in the hardware landscape (e.g., large DRAM capacities, fast NVMe SSDs, high-performance networks) and an increasing gap between what modern hardware is theoretically capable of and what real-world systems achieve. Currently, database architects have to rely on implicit assumptions (e.g., networking is the performance bottleneck) when making architectural decisions (e.g., compress data before sending it over the network). As illustrated in Figure 1, our cost-based approach enables rigorous, data-driven decision making, which will help bringing systems closer to the limits of the available hardware – in particular when evolving the system over time.

The rest of the paper is structured as follows: Section 2 covers important background and related work on analytical query processing in the cloud. In Section 3, we outline how a cost-optimality model can be used to improve cost efficiency by describing several specific applications. Section 4 then presents a simple but useful performance model for OLAP in the cloud based on intuitive rules (rather than black-box machine learning). We conclude in Section 5.

2 BACKGROUND AND RELATED WORK

Cloud SaaS data warehouses like Redshift for AWS [11], Snowflake for AWS, Azure, or GCP [9], and AnalyticDB for AlibabaCloud [24] allow querying large data sets without having to install software or procure physical hardware. With Redshift, customers have four main hardware options [1]: dc2.8xlarge (32 vCPUs, 244 GB RAM, 2.56 TB SSD), dc2.large (a sixteenth of dc2.8xlarge), ra3.16xlarge (48 vCPUs, 384 GB RAM, unspecified SSD cache size for S3), and ra3.4xlarge (a fourth of ra3.16xlarge). The instances can be combined to homogeneous clusters of up to 128 nodes, and Amazon recommends the newer ra3 class, which due to S3 support improves the separation between storage and compute. Some important resources (e.g., network speed) are unspecified and the instances are only available with Redshift, but not for general EC2 workloads.

Snowflake customers only have a single knob for influencing the hardware configuration: the "warehouse size" controls the number of servers and can be set to values between 1 ("X-Small") and 128 ("4X-Large"). The cost scales linearly with the number of servers. Snowflake does not specify the hardware configuration. However, performance debugging information suggests [2] that, on EC2, Snowflake currently relies on relatively small c5d.2xlarge instances (8 vCPUs, 16 GB DRAM, one 200 GB NVMe SSD).

Both Redshift and Snowflake support scale out and are moving to a design where the local SSD is used as a cache for the data stored on S3. Snowflake invests more in compute, Redshift more in DRAM. Snowflake relies on small instances and scale out, whereas Redshift also offers larger instance types making scale out less important. Overall, the hardware options for both systems are fairly limited in comparison to the instance types available on EC2. Both systems strive to be elastic and to separate storage from compute, though Snowflake still leads in terms of elasticity: its lazy caching architecture [22] makes resizing instant, and, by default, all instances automatically shut down after 15 minutes of inactivity.

The cost structure of similar DB-as-a-Service products has been analyzed in other work, generally concluding that the current offerings make it hard for customers to choose cost-optimal configurations. Floratu et al. [10] found that, in the long run, renting commercial systems with an hourly license fee is often cheaper than free alternatives due to their superior performance. To unburden users from such counter-intuitive choices, other work suggests presenting customers pre-packaged SLAs based on price and performance predictions that rely on workload statistics and query optimizer cost models [19]. Although the underlying motivations are similar, this work focuses on database engineers building systems that run on IaaS offerings, rather than customers of managed systems. Consequently, our approach differs in that we explicitly model the workload and how it relates to hardware, while simultaneously abstracting from any specific implementation.

There is also a second class of systems with a radically different pricing model that is arguably even closer to the SaaS spirit in the cloud. Instead of paying for clusters (and having to pick their size), services like Google BigQuery or Amazon Athena allow users to only pay per data access. In both BigQuery and Athena, scanning 1 TB of data costs $5, regardless of the query. It is not known whether these systems implement dynamic hardware selection.

There is prior work on hardware selection in the cloud focusing on how to automatically tune the hardware configuration for existing database systems [8]. One representative, recent example is *OPTIMUSCLOUD* [16], which optimizes performance per dollar for Cassandra and Redis by tuning software configuration and determining hardware instances. The basic approach is to treat the DBMS as a black box and predict performance changes for different hardware and software configurations based on earlier executions. Another approach is to rely on reinforcement learning [17]. The goal of this paper, in contrast, is to develop a general model for analytical query processing in the cloud that not only recommends hardware but also provides intuition on system architecture.

There are some existing formulas and online tools for selecting cluster sizes and configuration options for databases based on various workload metrics. For example, AWS published rule-of-thumb formulas for cluster sizing and resizing in Redshift [4]. Microsoft started providing a performance measurement tool for estimating the minimum managed Azure SQL database size when migrating from the on-premise version, that superseded an older, independently-developed online tool [5, 6]. These solutions focus on specific, managed database products, limiting their utility to database developers and people running non-managed databases.

The amount of prior work on OLAP in the cloud is large and we can only mention some of it. Snowflake recently published a workload trace containing performance statistics for all queries over a two-week period in 2018 [22] that was instrumental in understanding and modeling OLAP workloads. Tan et al. [21] experimentally compare OLAP systems (including Redshift) on EC2. There has been work on analytics using spot instances [13, 14], a possibility we mention in this paper, and using serverless infrastructure [18, 20], which we do not consider here. Workload-dependent hardware selection has also been suggested by Wei et al. [23].

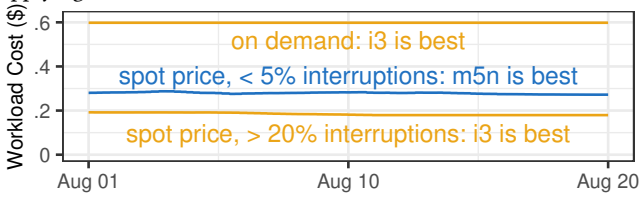
3 TOWARDS COST-OPTIMAL OLAP

This paper is based on two key ideas. The first is that in a heterogeneous cloud hardware landscape the goal must be to minimize workload cost rather than runtime. This idea is neither original nor, we believe, controversial. However, the second point may require some justification: We argue that it is useful to have a performance model for a hypothetical, idealized system that is capable of exploiting all hardware resources available in the cloud. The system we model can, for example, take full advantage of 100 Gbit networks and large arrays of NVMe SSDs. Since until recently such hardware was not available [12], most existing systems fall short of this assumption. One may therefore reasonably ask what the point of modeling a hypothetical system of this caliber is. We argue that precisely because such a system does not yet exist, having a model that shows the shortcomings of existing systems is useful:

Cost Optimality As A Benchmark For New Systems. While building a new cloud-native query processing engine that performs as well as our model is challenging, it is not impossible. Indeed, our model only assumes well-understood database concepts like caching, distributed query processing, and hybrid hash join-like operators. Thus, the hurdles are mainly a matter of engineering and systems building. We believe that, during system design and development, our cost optimality model is a useful benchmark against which a new system can be compared.

Evidence-Based Performance Feature Prioritization. Our model includes performance optimizations like multi-layer instance-local caching and distributed query processing. Disabling these features in the model would allow system architects to quantify how large the performance and cost benefit is. This enables evidence-based decisions about prioritizing performance features.

Market Pricing. Every cost calculation in this paper is done using *on demand* prices, which are stable but also fairly high. For elastic systems and fluctuating workloads, a potentially more economic alternative are *spot instances*, which are often only 20% to 30% of the on demand cost but fluctuate on an hourly basis. Our cost-based approach makes it easy to exploit spot instances: we simply have to periodically (e.g., every hour) re-run our model using the current spot market prices, and migrate to a new instance type if beneficial. The drawback when using spot instances is that they can be shut down at very short notice. The probability of this happening depends on the instance type, and the numbers are published by Amazon in the form of *interruption frequencies*. An interruption frequency of five percent for an m5n instance, for example, means that five percent of m5n spot instances were interrupted in the last month to regain capacity [3]. Depending on the maximum acceptable interruption frequency, different cost savings can be achieved. We measured this by gathering spot market data for three weeks and applying the model to it. The result for one workload is shown here:



Occasionally, we observed that a particular spot instance is not available (i.e., launching that instance simply fails) even though the market price is low. In this case, if the instance type chosen by the model is not available, we can simply switch to the second best option, which usually is only slightly worse.

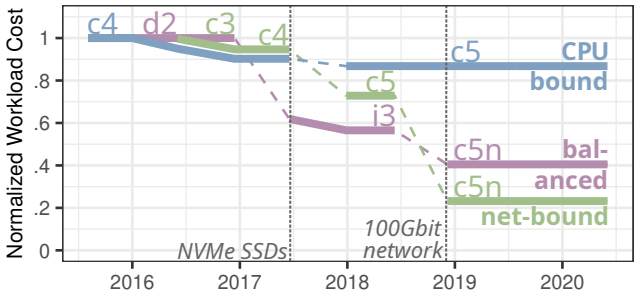


Figure 2: Opt. instance for different workloads over 5 years

Hardware/Software Co-Evolution. The available hardware in the cloud is not static, but changes over time. Our model allows reacting to new hardware opportunities by recomputing the model with the new hardware data. We can illustrate this idea by applying our model to hardware configurations and prices from the past 5 years. Figure 2 retroactively shows how the costs for three different workloads (CPU-bound, balanced, and network-bound) developed according to the model. Since 2015, there have been two major relevant changes in the EC2 hardware landscape. The first was the introduction of fast NVMe SSDs in mid 2017. This had a significant impact on the balanced workload, which moved from an instance with 20 disks (d2) to an instance with 8 NVMe SSDs (i3) – almost halving workload cost. The second major change was the introduction of 100 Gbit network instances at the end of 2018, which had even larger consequences: the cost of the network-bound workload dropped to a quarter of the initial cost (and even the balanced workload switched away from i3 to c5n). The CPU-bound workload, in contrast, did not see large gains in our 5-year period – which we find surprising given the rising number of CPU cores in commodity servers. This historical example illustrates how our model can be used to react to changes in the hardware landscape. When a new instance type with novel hardware is announced, our model can be used to quantify how large the benefit of switching would be. This benefit may then be compared with the engineering effort required to exploit the new hardware.

4 MODELING OPTIMAL OLAP IN THE CLOUD

One important component for achieving cost optimality is picking the best hardware for a particular query workload. Hardware selection is a challenging problem because any hardware resource (CPU, DRAM, SSD, network) can have a substantial impact on query performance – depending on the workload, any resource can be the bottleneck. One approach would be to choose a balanced hardware configuration, i.e., one where the budget is spent roughly evenly across all resources. However, for some workloads using a balanced configuration may be highly wasteful: why spend money on expensive SSDs, for example, if the data fits into RAM? Besides considering the available instances and prices, it is therefore crucial to model not just the hardware but also the workload.

In the following, after stating some basic assumptions, we present a sequence of models that predict how expensive a particular workload is on some given instance. For expository reasons, we start with a very basic model and gradually augment it to make it more

Table 1: Selection of EC2 instances (June 2020, us-east-1)

inst.	cores		DRAM		SSD		network		cost ↑
name	#	/$	GB	/$	TB	/$	Gbit/s	/$	$/h
c5n.18	36	9.3	192	49.4	-	-	100	**25.7**	3.89
c5.24	48	**11.8**	192	47.1	-	-	25	6.1	4.08
c5d.24	48	10.4	192	41.7	4×0.9	0.78	25	5.4	4.61
m5.24	48	10.4	384	83.3	-	-	25	5.4	4.61
i3.16	32	6.4	488	97.8	8×1.9	3.04	25	5.0	4.99
m5d.24	48	8.8	384	70.8	4×0.9	0.66	25	4.6	5.42
m5n.24	48	8.4	384	67.2	-	-	100	17.5	5.71
r5.24	48	7.9	768	127.0	-	-	25	4.1	6.05
m5dn.24	48	7.4	384	58.8	4×0.9	0.55	100	15.3	6.53
r5d.24	48	6.9	768	111.1	4×0.9	0.52	25	3.6	6.91
r5n.24	48	6.7	768	107.4	-	-	100	14.0	7.15
r5dn.24	48	6.0	768	95.8	4×0.9	0.45	100	12.5	8.02
i3en.24	48	4.4	768	70.8	8×7.5	**5.53**	100	9.2	10.85
x1e.32	64	2.4	3,904	**146.3**	2×1.9	0.14	25	0.9	26.69

realistic. We use the AWS EC2 hardware data for Linux instances from `https://ec2instances.info` as of June 2, 2020 in us-east-1 (North Virginia). A selection of the instances is shown in Table 1. Note that our model-based approach would also be applicable to other public clouds – it would even allow us to compare costs across different clouds.

4.1 Scope and Assumptions

Throughout the paper, we focus on analytical query processing rather than, for example, transactional or machine learning workloads. This allows us to focus on throughput and ignore latency (e.g., when reading data from storage). To keep things manageable, we also ignore instances with GPUs and FPGAs, which are available in the cloud but are still not widely used for OLAP.

We assume that the primary persistent storage medium is a distributed file storage service like Amazon S3[1]. S3 is a solid foundation for analytical data-intensive systems as it provides reliable persistence and high bandwidth. S3 is also much cheaper than alternatives like instance storage or EBS [21]. Assuming that data is always stored on S3 has a number of implications for our model. First, we can ignore the S3 storage cost because it is equal for all hardware configurations and therefore does not affect hardware selection. Furthermore, S3 does not charge for bandwidth and per-request costs are negligible in comparison with instance costs if we access large data chunks (e.g., 100 MB or more). Finally, since S3 is connected through the network, we assume that the S3 access speed is equal to 80% of the network speed, which is the maximum bandwidth from S3 we measured with 100 Gbit instances.

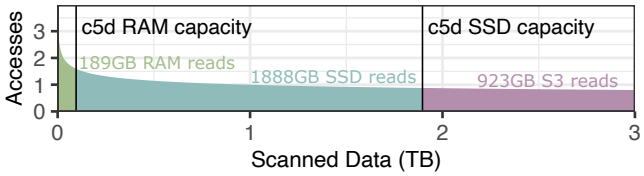
4.2 Basic Model (M1)

We start modeling a query workload using only two variables: **`CPU hours`** and **`scanned data`**. Since we initially assume data is always scanned from S3, query efficiency only depends on the network speed and the number of CPU cores of an instance.

Our goal is to calculate the workload cost on a particular instance. To do this, we first have to calculate the execution time. Let us illustrate this calculation with a workload of 1 CPU hour and 10 GB scanned running on instance c5.24, which has 48 cores and a 25 Gbit network card. Executing 1 CPU hour with 48 cores takes $1h/48 = 75s$ seconds and reading 10 GB over a 25 Gbit network takes $10GB/(25Gbit/s * 80\%) = 4s$. We assume that the CPU and network phases are consecutive, which means that the total execution time for the example workload would be $75s + 4s = 79s$. Finally, we have to multiply the execution time with the instance cost, resulting in a workload cost of $75s * \$4.08/h = \$0.085$. Using this approach, we can calculate how expensive a workload would be on a particular instance type. In the basic model, only CPU and network speed matter and instances with large DRAM (e.g., m5dn) or SSD (e.g., i3en) capacities are not beneficial. As Table 1 suggests, either c5 or c5n is the cheapest instance depending on the ratio between CPU hours and scanned data in this model.

4.3 Caching and Materializing Operators (M2)

We next incorporate data caching: instead of reading all data from S3 and therefore the network, we buffer data on DRAM and SSD. We use half of the available DRAM and SSD capacities for caching scanned data (the rest will be used for materializing intermediate results). Since real-world data accesses are skewed (e.g., some columns are accessed more frequently than others), we add a **`cache skew`** variable based on the Zipf distribution to the model. Let us illustrate the impact of this variable using an example, where 3 TB are scanned on a c5d.24 instance with a skew of 0.2:



The amount of data accessed for each level depends not just on the skew setting but also the instance type (numbers in GB):

inst.	no cache skew			cache skew=0.2			cache skew=0.9		
name	RAM	SSD	S3	RAM	SSD	S3	RAM	SSD	S3
c5n.18	96	-	2,904	189	-	2,811	1,491	-	1,509
c5d.24	96	1,800	1,104	189	1,888	923	1,491	1,276	233
i3en.24	384	2,616	0	577	2423	0	2,034	966	0

DRAM and instance storage are not only used to scan input data, but also for query processing. Complex queries often materialize parts of the scanned data, e.g., during joins or aggregations. Since we want to be able to execute arbitrarily large queries on any instance, it must be possible to temporarily materialize large data sets. As with data storage, we model a three-layer hierarchy consisting of DRAM, SSD (if available on that instance), and S3. Thus, like in Snowflake [22], for large queries, S3 acts as ephemeral storage of unlimited capacity.

To model materialization, we introduce the **`materialization fraction`** variable that determines how much of the scanned data has to be materialized. For example, with 1 TB scanned and a materialization fraction of 0.1, 100 GB of data has to be materialized. In most workloads, the fraction is well below 1 because scanned data is often filtered before being materialized. For example, the average materialization fraction across all Snowflake customers is 0.3 – though there is substantial variability across customers.

It is not enough to know the total volume of materialized data, but it has to be broken down to DRAM, SSD, and S3. In practice, most

[1] Microsoft (Azure Blob Storage) and Google (Cloud Storage) offer very similar services.

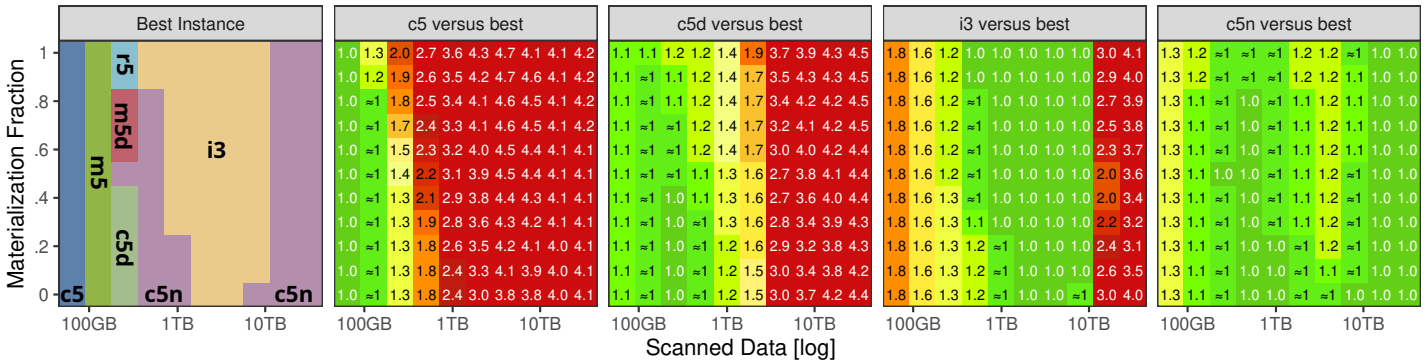


Figure 3: Model M2 with 1 CPU hour and 0.001 cache and mat. skew, illustrating that the best instance depends on the workload

queries materialize little data and DRAM is large enough for them. Fewer queries require spooling to SSD, and even fewer to S3. In the Snowflake workload, for example, 81% of all read-only queries are materialized to DRAM, 18% to SSD, and 0.8% to S3. We model this by introducing the `materialization skew` variable. The effect of materialization skew is similar to cache skew, i.e., it shifts accesses from S3 to SSD and main memory. High materialization skew values model in-memory query processing, and low values out-of-memory processing using SSD and/or S3.

Caching and materialization cause instance types with larger DRAM and SSD capacities to become more attractive. To illustrate this effect we vary the scanned data and materialization fraction, but keep the CPU hours (1) and both skew parameters (0.001) constant. The left-most plot in Figure 3 shows the cheapest instance for 110 different workloads. For low amounts of scanned data (<100 GB), the workload is CPU bound, and thus c5 is the cheapest instance. For high amounts of scanned data (>10 TB), c5n wins because the data sets are so large that caching does not help and only fast network matters. But unlike the basic model (M1), in between these two extremes, instances with larger DRAM capacities (m5 and r5) and SSDs (c5d and i3) are optimal. Interestingly, between the area where c5d and i3 win, the SSD-less c5n instance is cheapest. This is because instance storage in EC2 is only available in certain granularities and in this area c5d has too little capacity while i3 is too expensive.

Let us now look at the cost differences between instances rather than just the cheapest instance. The four plots on the right of Figure 3 show the normalized workload costs (relative to the optimal instance) for four selected instances. We see that in some settings, c5, c5d, and i3 have more than four times the cost of the optimal instance, even though these instances are optimal in some cases themselves. This shows that choosing a good instance can have a large impact and that the choice depends on the workload. Another finding is that c5n is remarkably good across all configurations, being at most 1.26× more expensive in the worst case. c5n has 100 Gbit networking but little DRAM and no instance storage – therefore it relies heavily on S3. Our model assumes that it is possible to read and write with 10 GB/s from S3, which is only surpassed by instances with 8 NVMe SSDs, which are much more expensive than c5n (cf. Table 1). This makes c5n highly attractive for large-scale OLAP in the current EC2 pricing structure[2]. 100 Gbit networking has only recently been introduced to EC2 and seemingly led to a tectonic shift in the hardware landscape, as depicted in Figure 2.

4.4 Scale Out (M3)

We next model scale out, i.e., the capability of using more than one node for query execution. We assume that cached data is horizontally partitioned across nodes. Thus, in effect, the total effective cache size grows linearly with the number of nodes. While, in principle, most resources grow linearly with the number of nodes, in practice, achieving perfect scalability in a distributed setting is unrealistic – in particular when many nodes are involved. We therefore introduce the `scalability fraction` variable that determines how much of the workload time can be scaled. The variable models how well the system scales and, using Amdahl's law, we can compute the speedup for the workload. For example, a scalability fraction of 0.9 means that the maximum speedup is 10 (even with an infinite number of instances), while a scalability fraction of 1 models perfect scalability.

Another downside of distributed query processing is that it requires the nodes to exchange data with each other through the network, for example, when joining or aggregating. The volume of exchanged data for analytical queries is very similar to the amount of data that needs to be materialized (Materialization is usually caused by operators like join and aggregation that also require exchanging the same amount of data). Instead of introducing a new variable, we can therefore reuse the materialization fraction variable from the previous version of the model to compute the volume of the required network traffic. Thus, if a workload is executed on multiple instances, we add a networking phase for exchanging the materialized data. If only a single instance is used, this phase is not necessary, resulting in a (realistic) discontinuity between distributed and single-node query processing.

Figure 4 illustrates how scaling out (to at most 128 nodes) using a scalability fraction of 0.95 affects execution time and cost. Depending on the instance type and count, we observe differences of almost two orders of magnitude. In the lower-left part of the figure we see the Pareto frontier consisting of instance types c5n, r5n, and m5n. The model switches from 24 c5n to 26 r5n nodes and from 37 r5n to 52 m5n nodes, exploiting their larger DRAM capacities, which only become significant with larger node counts in aggregation. However, the figure also shows that with a scalability fraction

[2]Recall that S3 itself is cheap because we assume data is on S3 anyway (for fault tolerance) and that request sizes are large enough to make per-request costs negligible. Also note that S3 and the network are resources shared between multiple users with potentially unstable performance, but we cannot easily model this. For these reasons, our model makes S3 look very good.

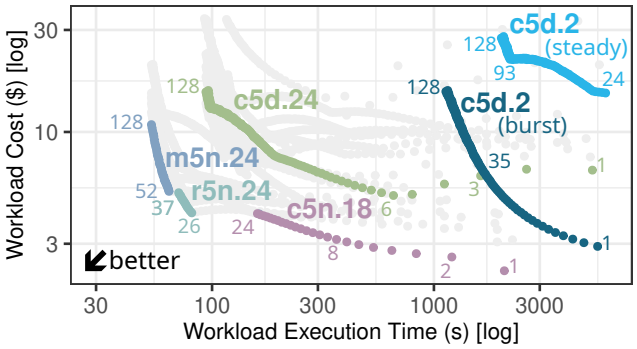


Figure 4: Model M3 with 5 CPU hours, 10 TB scan, 0.001 skew, 0.3 materialization fraction, and 0.95 scalability fraction

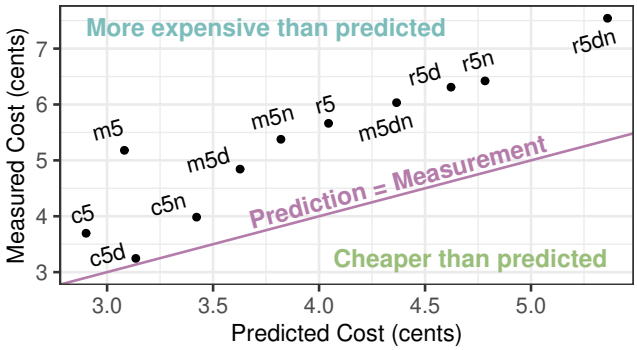


Figure 5: Prototype measurement vs. prediction on a 100 GB aggregation query

of 0.95, the workload cost rises with larger node counts because scalability decreases – creating a tradeoff between cost and time.

All results presented so far rely on full instances. However, as discussed in Section 2, many systems use smaller instance slices. c5d.2, for example, which Snowflake seems to be use, has a twelfth of the resources and price of a full c5d.24 instance. One exception to this rule is network bandwidth: For small slices like c5n.4xlarge and below, AWS specifies the network bandwidth as "Up to 25 Gbit" and we have indeed temporarily observed close to 25 Gbit bandwidth. However, this seemingly free bandwidth is only available for a limited duration – after some time the bandwidth approaches the payed-for slice fraction. Since we model the steady state, smaller slices do not appear beneficial: a full instance will always have a lower or equal workload cost and lower runtime than any of its slices because we assume imperfect scalability when scaling out. The *c5d.2 (steady)* curve in Figure 4, with its much higher workload cost, shows this clearly. However, slices can certainly be useful for bursty workloads, as the *c5d.2 (burst)* curve in the figure shows.

4.5 Discussion and Preliminary Evaluation

Let us discuss some possible objections to the model. With only six variables, our model is quite simple, and indeed we do not claim to accurately predict the execution time of any existing system. Trying to model the performance behavior of a real system would make the model and its predictions almost as hard to understand as the modeled system. Our goal is to have a robust model that is roughly correct and easy to reason about, rather than a complex model that tries to be completely accurate, which is probably unattainable anyway. The simplicity of the model also makes it quick to evaluate: to determine the best instance, we exhaustively enumerate all instance configurations for the given workload.

The model is based on a number of implicit assumptions that we would like to justify. We assume that CPU work can be perfectly parallelized on a single node (though we leave some slack by ignoring hyperthreads). This is based on our experience that such a scalability is indeed achievable on systems below 100 cores [15]. In a distributed setting, on the other hand, achieving perfect scalability is more difficult and the number of nodes is unbounded, which is why we chose to introduce the scalability fraction variable. We further assume that the networking, storage, and CPU phases are consecutive rather than overlapping (we use the sum rather than the maximum of the phase durations). While real systems may try to overlap these phases, queries with a significant networking or I/O phase will likely result in more CPU load as well, which is why we chose to be more conservative in this case.

Finally, the assumption that all data is (also) stored on S3 is based on the fact that S3 is much cheaper than instance storage: S3 costs less than $25 per TB per month, while the same would cost $130 on an i3en SSD instance and $83 on a d2 disk instance. A design primarily relying on instance storage is therefore rarely (and even then only marginally) beneficial in comparison with our S3 plus caching approach.

To show the viability of the model, we built a prototype query engine that can run simple aggregation queries. It is capable of loading data from S3, caching it in DRAM and on local SSDs if available, and using DRAM, SSDs, and S3 as storage for intermediary results. The model variables can be set based on the tested relation size and the number of unique values in the group column. We executed this engine on a variety of EC2 instances and measured the execution cost. The expected behavior is that the measured costs correlate with the model predictions, but are worse by a near-constant factor since the predictions target an ideal system. This behavior can be seen in Figure 5 for an aggregation query that scans 100 GB of data from S3, measured on 11 different EC2 instances.

5 SUMMARY AND FUTURE WORK

For cloud-native database systems, efficiency and cost effectiveness are closely related. This paper proposes an intuitive model for analytical query processing in the cloud that estimates the runtime and cost of a workload on a particular hardware instance. The most important applications of the model are making data-driven architectural design decisions, finding performance bottlenecks, and evolving data management systems towards the goal of cost-optimal query processing in the cloud. An implementation of the model is provided at `https://github.com/maxi-k/costoptimal-model`. Additionally, an interactive web-based tool for exploring the model is available at `https://maxi-k.shinyapps.io/costoptimal/`.

Finally, it has not escaped our notice that our economic approach can be applied to other areas besides analytical database query processing workloads. The training of machine learning models, for example, could be optimized with a similar methodology using a model that focuses on floating point calculations. Exploring our cost-based approach in other domains would be an interesting avenue for future work.

Acknowledgments. We thank Amazon for an *AWS Cloud Credits for Research* grant.

REFERENCES

[1] August 11, 2020. https://web.archive.org/web/20200811175333/https://aws.amazon.com/redshift/pricing/.

[2] August 11, 2020. https://stackoverflow.com/questions/58973007/what-are-the-specifications-of-a-snowflake-server.

[3] August 11, 2020. https://aws.amazon.com/de/ec2/spot/instance-advisor/.

[4] February 16, 2021. https://d1.awsstatic.com/whitepapers/Size-Cloud-Data-Warehouse-on-AWS.pdf.

[5] February 16, 2021. https://dtucalculator.azurewebsites.net/.

[6] February 16, 2021. https://docs.microsoft.com/en-us/sql/dma/dma-sku-recommend-sql-db.

[7] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing using RDMA. In *SIGMOD*.

[8] Muhammad Bilal, Marco Serafini, Marco Canini, and Rodrigo Rodrigues. 2020. Do the Best Cloud Configurations Grow on Trees? An Experimental Evaluation of Black Box Algorithms for Optimizing Cloud Workloads. *PVLDB* 13, 11 (2020).

[9] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*.

[10] Avrilia Floratou, Jignesh M. Patel, Willis Lang, and Alan Halverson. 2011. When Free Is Not Really Free: What Does It Cost to Run a Database Workload in the Cloud?. In *TPCTC (Lecture Notes in Computer Science, Vol. 7144)*. Springer.

[11] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD*.

[12] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[13] Dalia Kaulakiene, Christian Thomsen, Torben Bach Pedersen, Ugur Çetintemel, and Tim Kraska. 2015. SpotADAPT: Spot-Aware (re-)Deployment of Analytical Processing Tasks on Amazon EC2. In *DOLAP*.

[14] Tim Kraska, Elkhan Dadashov, and Carsten Binnig. 2017. Spotlytics: How to Use Cloud Market Places for Analytics?. In *BTW*.

[15] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*.

[16] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2020. OPTIMUSCLOUD: Heterogeneous Configuration Optimization for Distributed Databases in the Cloud. In *USENIX ATC*.

[17] Ryan Marcus and Olga Papaemmanouil. 2017. Releasing Cloud Databases for the Chains of Performance Prediction Models. In *CIDR*.

[18] Ingo Müller, Renato Marroquin, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD*.

[19] Jennifer Ortiz, Victor Teixeira de Almeida, and Magdalena Balazinska. 2015. Changing the Face of Database Cloud Services with Personalized Service Level Agreements. In *CIDR*.

[20] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD*.

[21] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David J. DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. *PVLDB* 12, 12 (2019).

[22] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*.

[23] Lei Wei, Chuan Heng Foh, Bingsheng He, and Jianfei Cai. 2018. Towards Efficient Resource Allocation for Heterogeneous Workloads in IaaS Clouds. *IEEE Trans. Cloud Comput.* 6, 1 (2018).

[24] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP Database System at Alibaba Cloud. *PVLDB* 12, 12 (2019).