Section 4: Week 8: Enhance a Distributed System Architecture

Nate Bachmeier

TIM-8120: Distributed Systems

November 10th, 2019

North Central University

# Enhance a Distributed System Architecture

Contoso Retail and Manufacturing is similar to many large enterprises in their need to manage and support heterogeneous networks that span multiple data centers and branch offices. Their workloads include standard services such as authentication, name resolution, and distributed storage system, along with proprietary systems. This topology naturally steers them towards distributed system designs, which come with their own sets of challenges, like balancing availability, reliability, and performance. Many of these challenges are NP-hard, meaning that the precise answers are complex to derive but easy to verify. Therefore they require efficient approximation algorithms or revolutionary changes in approach. Other aspects of the system need architectural patterns that enable Recovery-Oriented Programming (ROP)*.*

# What is Recovery-Oriented Programming

Fault-tolerant systems do not happen by accident and are the product of specific architectural patterns (Zhao, 2014). Zhao describes several variations of logging, checkpointing, isolation, replication strategies, along with implications such as group communication and consensus. A checkpoint must contain sufficient context for a recovery operation to continue the request at a later time. An isolation strategy will partition a logical resource into disjoined physical units so that the blast radius of a fault is manageable. Replication of both services and data provides a mechanism to absorb load and horizontally distribute it.

## Using Checkpointing

In Figure 2, an actor performs actions A-C and D-F with the fundamental difference being D-F creates state checkpoints between the operations. Consider the difference in recovery responsibility of a fail-stop between B-C and E-F. In the case of B-C, the error needs to be trapped and retried by the actor. Then contrast that with E-F where the responsibility lies with E. While it may be sufficient for specific scenarios to rely on the actor to make these recovery decisions, they become proportionally complex relative to the number of services involved.

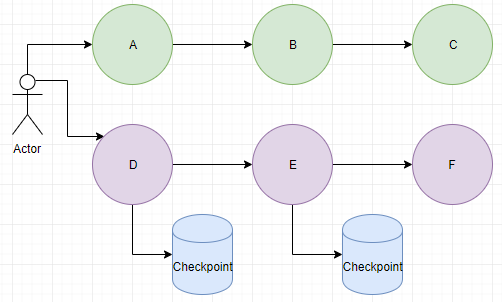


Figure 2: Checkpointing

## Using Service Partitioning and Replication

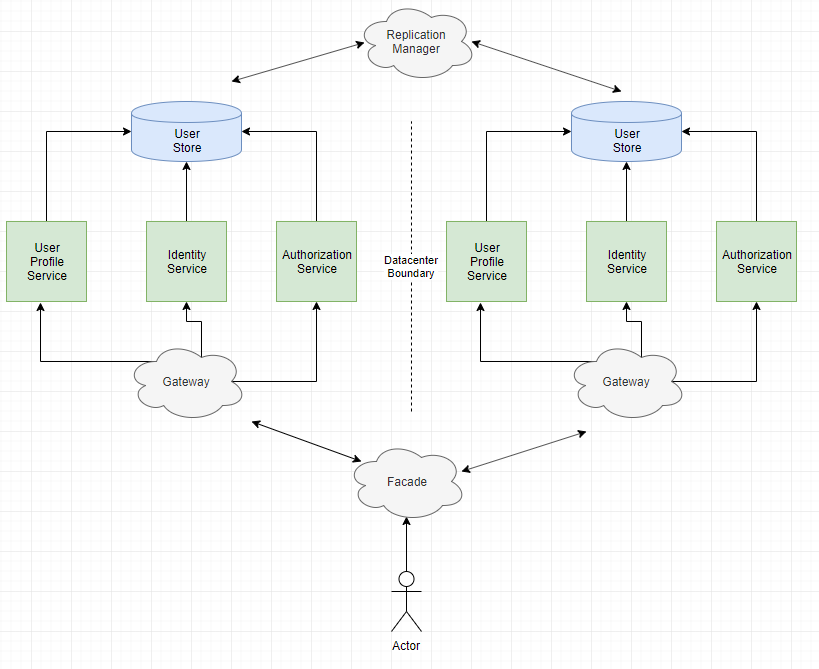


Figure 3: Isolation and Replication

To ensure a collection of user services are highly available, the system designers can leverage service isolation and replication strategies (see Figure 3). This approach starts with decomposing the different domains of user management into separate micro-services, such as user profiles from authorization policies. Containerization and other virtualization technologies can create security barriers between each micro-service instance, such that the underlying physical hardware is sharable while minimizing additional risk (Qu, 2018) (Ahmad, Naveed, & Noda, 2018).

## Using Rejuvenation Policy

Load balancers and service orchestration engines can manage the lifecycle of these micro-services, through elastic provisioning and application of software rejuvenation policies (Yang, Min, Yang, & Li, 2013). For instance, a memory leak exists in the user profile service that eventually results in a fail-stop. The business cannot resolve the issue as they lack the expertise, due to the component using closed source software. Administrators mitigate the problem by provisioning multiple instances of the profile service, along with monitoring, and automation that recycles instances above X% committed memory. The micro-reboot of the service instance can then become managed and controlled. Instead of hard stopping, the automation can first take it out of the load balancer target group resulting in a natural drain of requests. Then after a specified duration or heuristic, the instance can be safely cycled.

## Using Datacenter Fail-over

Partitioning a service into multiple micro-services and then provisioning numerous copies of them, reduces the impact of an individual unit encountering a fail-stop scenario. While the strategy provides localized protections, they are ineffective against data center outages. Based on reporting from multiple news sources, it seems a significant disruption in a Public Cloud Service Provider (CSP) happens every six to twelve months (McGlaun, 2018) (Targett, 2019) (RIQ News Desk, 2016). Mitigation of these outages requires geo-replication of both services and data to multiple disjoined physical locations. For instance, the example user service could physically reside in Seattle and Boston. Then the actor can call into a service façade and rely on their traffic flowing to the closest availability zone (Qu, 2018).

# Enhance a System with Distributed Technologies

## Challenges of Eventual Consistency

Some argue that only strongly consistent reads and writes to data stores should be used (Liu, Arden, George, & Myers, 2017). However, this assertion is challenging in geo-replicated data stores, given the latency and performance penalty of remote synchronization. Multiple protocols exist for partitioning the data within these stores and coordinating distributed transactions across them, each with different scenario optimizations (Bharati & Attar, 2018) (Zhao, 2014) (Almeida & Baquero, 2013). Almeida and Baquero propose a solution that relies on Eventually Consistent Distributed Counters (ECDC), where each node periodically recalibrates with their peers. Bharati and Attar expand on these ideas through a survey of open source technologies that use asynchronous Paxos, strict two-phase commit, optimistic concurrency, and even graph-based workload aware replication.

## Maintaining Consistency across Heterogeneous Datastores

Significant amounts of literature exist around implementing replication patterns across homogeneous datastores, yet Software as a Service (SaaS) paradigms are pushing the industry towards “built for purpose” strategies that span heterogeneous solutions (Brice & Idziorek, 2018). This evolutionary step also makes logical sense, as distributed systems rely on different data structures to efficiently perform specific expert scenarios. Consider the difference between a search and graph store, where one enables *term* *indexing* across a collection of documents, and the other explores entity *relationships*. The data within them might be from the same domain, but their access patterns require different technologies.

To address these replication challenges, researchers have looked towards centralized orchestration services (see Figure 4) (Limon et al., 2018) (Venkatesan & Sridhar, 2015). Limon et al. propose a multi-agent Saga solution for issuing actions and compensations. Venkatesan et al. describe a system built on Apache Orchestration Director Engine (ODE) and Business Process Execution Language (BPEL). Both systems track progress through a hierarchial in-flight transaction store with each service request starting a new branch. As the leaf nodes report success or failures, the status propagates to parent branches. If a branch enters into the failure state and a compensation action exists, it will be involved to retry or rollback the action operation.

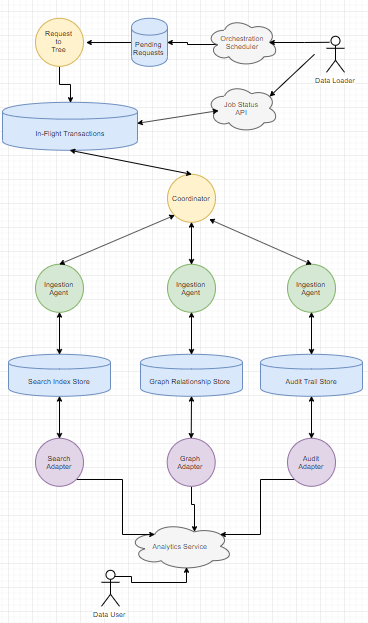


Figure 4: Multi-Agent Coordination

## Maintaining Consistency across Multi-Master Replicas

Contoso has a business requirement that its third-party product catalog is modifiable and queriable from a geo-aware replica. This need expands on the multi-agent solution to include Multi-Master Full Replication (MMFR)(see Figure 5). MMFR systems come with a unique set of challenges for maintaining reliability and performance. For instance, the need for more real-time updates increases the amount of communication and forces an asynchronous model (Macdeo et al., 2008). There are also additional complexities for traditional relational stores as each site must maintain enormous changelogs to rollback transactions (Filip, Vasar, & Robu, 2009). Amazon’s Dynamo service removes this constraint by broadcasting per item changes that using an array of vector clocks for versioning (Vogels, 2007). When a vector clock versioned item arrives, the local node attempts to prune competing instances and then allows the application to determine the final merge operation (e.g., last write wins or most recent not null value).

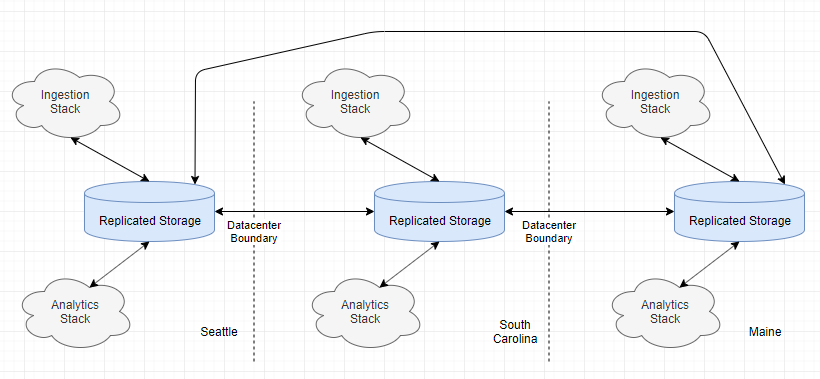


Figure 5: Multi-Master Full Replication Level 1

In Figure 6, the component level design shows the replication subsystem that exists between two geo-replicated environments. It relies on fully asynchronous communication by placing tasks inside of queues to signal that additional processing is needed (Alappatt, 2018). The first sink is the Pending Requests Queue (PRQ), which contains the requestor’s payload, version number, and security context. The Storage Router Service is a collection of elastic micro-services that scales proportionally to the queue depth (Sachs, Kounev, & Buchmann, 2013), and fetches any pending requests that are waiting for translation from the virtual object name into a physical storage node. The result of the storage operation, such as get data value or update a record, is given to the caller. When a storage node has acknowledged a change, it sends a replication request for that item into the Write Buffer Queue (WBQ). These are multicast to all peers that appear in the Peer Replica List (PRL).

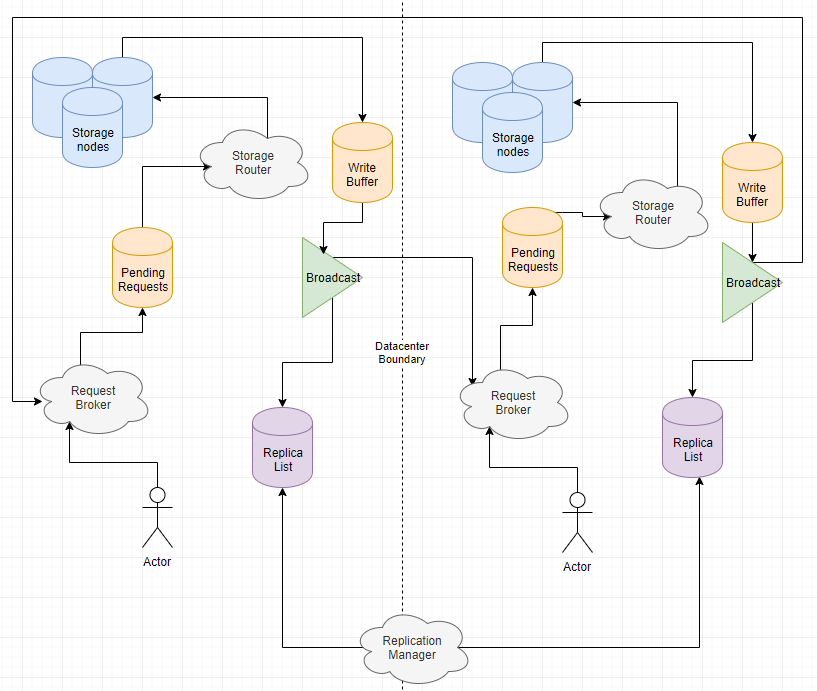


Figure 6: Multi-Master Full Replication Level 2

# Literature Review on the Influence of NP-Hard

## What is NP Complexity

If the solution to a decision problem takes *polynomial time* to solve, then it would have a complexity of *P*. These questions would include addition, sorting a list, and performing many procedural transformations. Other problems take *nondeterministic polynomial time* to solve and fall into the category of *NP*. If the solution to an NP question is verifiable in polynomial time, then it is *NP-hard*. Examples include hardware verification, multiprocessor scheduling, and even Super Mario Brothers (Mann, 2017) (Aloupis, Demaine, & Guo, 2012). When both the answer and verification take nondeterministic polynomial time, these challenges are said to be *NP-complete*.

According to Aloupis et al., if and only if, a decision problem can be proven to be as tough as an existing NP-hard problem, can it be considered NP-hard. A common strategy is to map the new challenge to the classic *Satisfaction Problem* (SAT). SAT is known to be NP-hard as its “combination of Boolean variables, negations, disjunctions, and conjunctions […] cannot be solved more efficiently than exhaustive search (2^n steps) (Mann, 2017, p. 76).” Consider a Mario level (see Figure 1), where the player needs to collect (*True*) and use (*False*) power-ups in a precise sequence of events to reach the end. A model of these decisions can be completely mapped to the SAT and therefore said with certainty, to solve Mario is at least equally hard to solving SAT.

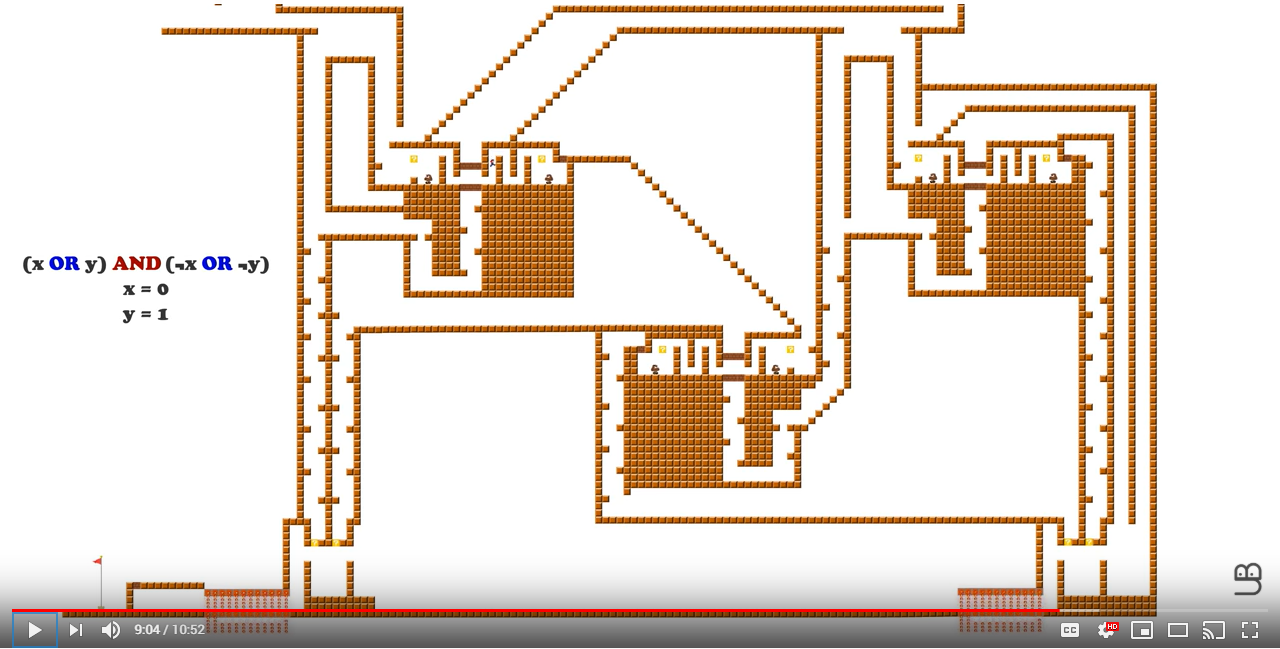


Figure 1 Super Mario as SAT (Undefined Behavior, 2019)

Mann states that there are many misconceptions around the notion of NP-hard and that literature frequently misrepresents the establishment of this assignment. Many scenarios, such as combinatorial enumerations, require exponential but finite time making them P. He also highlights that many problems *appear* similar, but can through changes to the model no longer be map precisely. These changes might include additional constraints to define the scope of the solution. Imagine the previous Mario level, where the player can walk through walls or use a ladder (oracle) to skip the maze and directly reach the end. Lastly, he cautions against the presumption that NP-hard can only solutions only exist through exhaustive search and heuristics. Many complex problems, such as bin packing, are addressable on a practical basis with a consistent selection strategy.

## Scheduling Jobs with Multitasking and Asymmetric Switching Costs (2017)

A core responsibility of many scheduling algorithms is to complete a collection of work on time. If the number of processors is finite, but the work items unbounded, the system will need to context switch and interrupt execution. Determining the total number of jobs that will be late is NP-Hard (Ho & Sum, 2017). Ho and Sum describe a particular set of scenarios that linear assignment can solve. These optimizations come with a few constraints, such as late jobs are given a lower priority than those that are likely to complete on time. They continue with mathematical proof that if the approximate runtimes and count of processes are known, the makespan and related properties take O(n^3) time.

While this strategy would not work for all scenarios, it applies to other real-world challenges. For instance, a rideshare service has space for eight riders per van, but most passengers travel in sets of one or two. How can they most efficiently pickup and dropoff both existing and new business, such that everyone is on time? Since we know the pickup and destination, the passenger’s approximate *runtime* is known. This data point enables the computation to prune multiple decision branches and reduce the time complexity to a more manageable size.

## A Constant Factor Approximation Algorithm for Unsplittable Flow on Paths (2011)

Some tasks cannot be interrupted and will use a fixed amount of a shared resource for a particular duration. This requirement “naturally occurs and so has studied under the names bandwidth allocation, admission control, interval packing, temporal knapsack, multicommodity demand flow, unsplittable flow problem, and scheduling with fixed start and end times (Bonsma, Schulz, & Wiese, 2011).” Other literature has extensively investigated the issue and identified many shortcuts such as (a) approximation heuristics and (b) applying constraints, such as requiring no bottlenecks. The researchers provide a general solution by first transforming the problem into a maximum weight independent set of rectangles problem, which has a complexity factor of .

These approximations apply to distributed systems, such as when tasks need to execution across a group of compute nodes. A scheduler process could bin pack these tasks, such that each batch was roughly the same size. Then these batches could be assigned to the different nodes, such that all resources and utilized at a consistent rate and their expected start and end times reliably derived.

## Reliability Calculation of P2P Streaming Systems with Bottleneck Links (2017)

A Peer-to-Peer (P2P) video streaming system consists of sources, sinks, and network links between them. Each link has a finite bandwidth that a probabilistic reliability rate of dropping packets. It is possible to model this environment as the “surviving probability of the reliability of [Graph] *G*, with respect to flow demand (Fujita, 2017, p. 1238).” Fujita argues that unless P = NP, generally determining the weighted sum of failure requires  steps. However, an optimization exists whereby creating a subgraph and then measuring the maximum flow. If the maximum flow holds, that confirms no bottlenecks are present in that segment. This shortcut can reduce the computational complexity to provided the count of links, and flow demand are constant. The edges that possess the most bottleneck will become the cut points for the subgraphs. This characteristic reduces the number of subgraphs that will need to measure, as similar link grades become clustered together.

Outside of P2P video streaming systems, this strategy applies to arbitrary networks that need to model constant flow demand across undependable links. While traffic ingested into the system is naturally bursty, a FIFO queue can first accept it, and then a metered process pulls these messages. Then through an understanding of the flow demand, an expert system could make decisions around provisioning new resources and related autoscaling remediations.

## The Complexity of the Infinity Replacement Problem in the Cyber Security Model (2017)

A network topology has many connected paths that eventually result in access to a given high valued asset. For instance, a web portal takes payment information and hands that off to a series of backend systems. If an attacker can gain access to the backend store, their prize is a list of credit card numbers. To exploit systems along this path requires the attacker to spend some amount of budget and effort. Therefore, a high-valued asset is flawlessly secure if and only if the cost to the attacker is infinity (Mukdasanit & Kantabutra, 2017). Previous efforts proved that modeling this problem was NP-complete, however by rooting the decision tree, it reduces to NP-hard. By further redefining the problem as finding the minimum edge, the researchers can answer the question in pseudo-polynomial time.

The fascinating aspect of this solution is that by clarifying and refining their specific objectives, these various researcher efforts could apply reasonable constraints to an otherwise arbitrary graph. This transformation of the problem occurred both with Mukdasanit et al. and their predecessors. That would suggest that there is potential value in fully understanding the business needs of a request before diving in with an implementation.

## Techniques and applications of computation slicing (2005)

When a distributed system encounters a fault (predicate), a mechanism needs to trigger a sequence of remediation steps. Detecting these predicates in the abstract is NP-complete unless decomposition of the *compute slice* occurs along with the classification of the subtask (Mittal & Garg, 2005). In the abstract, a fault monitoring system of *k* events across *n* processes would require k^n parameters. However, by using multiple snapshots (slices) of theses processes, the search space can be drastically only inspecting the changelog. In other words, if Alice calls Bob, there’s no reason to look at Charlie, the area of interest is between Alice and Bob. After the search space is constrained, efficient algorithms can query the call subgraph in polynomial time. Similar to other examples, the authors highlight the need to have well-defined objectives and not admit defeat because the most generic solution cannot be efficiently solved.

## Approximation Algorithms for Fault-Tolerant Facility Allocation (2013)

Contoso has a collection of sites, each (1) residing in a city, (2) has an operational cost, and (3) a connection cost to reach its peers. The enterprise’s goal is to introduce new facilities while minimizing the total operational expenditure (Shen & Xu, 2013). Shen and Xu reapply this classic operations research problem to the notion of *fault-tolerant facility allocation* (replica placement). The researchers describe how adding constraints, such as finite capacity at specific sites, has been shown to *increase* the complexity of this problem. They propose an iterative approximation algorithm called *inverse dual-fitting*, which uses attempts to minimize an objective function across multiple rounds. Fundamentally, they are using a greedy algorithm to make the best immediate choice and avoid the requirement to plan further ahead (NP-hard).

## Optimal bundles for sponsored search auctions via bracketing scheme (2019)

Selecting the most relevant sponsored content requires matching both the user-provided search term with the current advertiser bids for that space (Xia, Bu, & Gong, 2019). Xia et al. provide a formal proof that the industry-standard solution of bracketing bids is an NP-hard computation. Consider the scenario where (1) Alice bids 5$, Bob bids 2$, and Charlie bids 1$, and (2) the website only has space for two ads. Then the desired state would be to choose the top two advertisers and bill them the n+1 bid, such that Alice pays 2$, and Bob pays 1$.

Instead of creating and shifting brackets, the researchers propose a simple valuation matrix with discrete values of zero and one. Then a heuristic can be applied across k-signals to prune the search space to a finite list of potential matches. The purpose of the signals is to eliminate search terms that appear in multiple contexts (e.g., Java is a programming language and beverage). Across three test cases, they were able to prove the heuristic was approximately equivalent to the traditional bracketed solution.

The notion of applying filter masks to reduce the search space comes up in multiple distributed system scenarios. For instance, multi-attribute based resource discovery and load balancing rely on this idea (Lee, Keleher, & Sussman, 2014). Lazaro et al. propose that the attributes should use discrete values from a finite set, instead of only Boolean flags. They argue that this reduces the number of checks that need to occur, resulting in more efficient matching (Lazaro, Marques, Jorba, & Vilajosana, 2013).

## “Almost-stable” matchings in the Hospitals / Residents Problem with Couples (2017)

According to Manlove et al., when a graduate medical student does their residency, they must be matched with a hospital in their preferred list. If they have a colleague or significant other also performing a residency, they can request this person is geographically close to them. A process must compile these requirements and create assignments to the finite hospital positions. To further complicate matters, there are procedural rules that prevent specific scenarios, such as no one chose ABC Hospital (Manlove, McBride, & Trimble, 2017). The researchers provide two efficient algorithms to solve the problem in polynomial time. First, is an Integer Programming (IP) implementation that finds the maximum cardinality for each resident. Second, is a Constraint Programming (CP) solution that is 1.15 times faster than IP, and 8.14 times faster if pre-solving is enabled. Similar challenges exist within distributed systems, such as the case that a batch operation enters the system. When this occurs, a controller needs to assign tasks to specific and finite compute nodes. If two sub-tasks must share a data set, then an optimization exists to place both on the same node instead of replicating that data to another node. These requirements are analogous to the Hospital Residency Problem.

## On Complexity of Effective Data Granulation in Databases (2014)

An open problem in research is to devise an efficient *granulation* algorithm for columnar storage. Imagine business needs to determine total sales across the last month from a table that contains user\_id, date, item\_id, and price. The granulation of their query must (1) filter by the *date* column, (2) fetch the associated price value, (3) aggregate the values on each physical memory page, and then (4) sum the aggregation. Performing these steps is an NP-hard problem because the multiple layers of inclusion and exclusion are equivalent to SAT (Wróblewski & Kowalski, 2014). Wroblewski and Kowalski explain that the performance-critical point is to read a record once.

Based on their diagrams, it appears the fundamental issue is that a record contains a *directed* link to an offset in a per column file. However, if this structure is changed to be unidirectional and fully connected, then it should be possible to apply linear set theory for many common scenarios (see Figure 7). For instance, the original example could start at the sorted *date* column file, filter appropriately, and traverse the object pointers directly to the value offset in the price column file. These prices can be enumerated and inline incremented during the traversal.

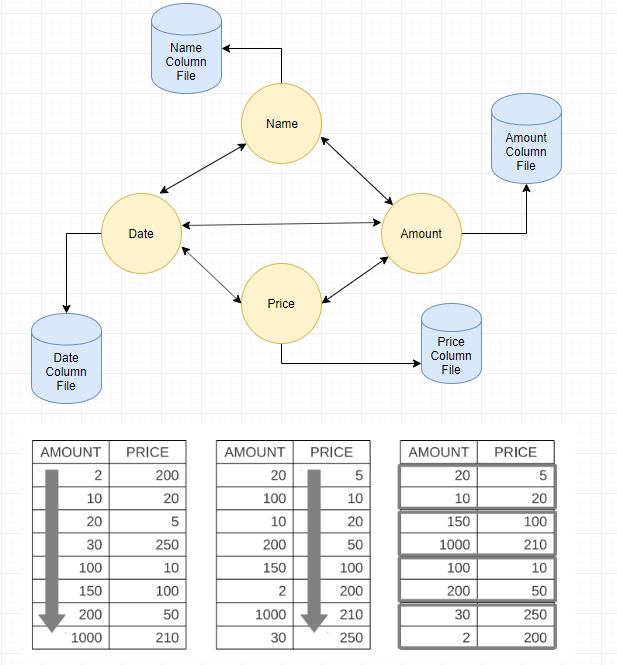


Figure 7: Graph vs. Row vs. Column Storage

## An improved algorithm for the MaxSAT problem (2019)

The (n, x)-Max SAT problem, contains *n* Boolean variables across *x* clauses, as is the defacto example of NP-hard. A naïve solution can enumerate all permutations and solve the puzzle in 2n steps. However, researchers have found strategies to remove terms and reduce the search space by applying discrete mathematics equivalency transforms (e.g., True & True & False *is* False) (Xu, Li, Wang, & Yang, 2019). Xu et al. extend this idea to include branching, which creates multiple parallel solvers, each seeded with different permutations of the *n-*input set. As various combinations are proven or rejected by individual clauses, then other pending branches can be pruned based using Constraint Programming (Manlove, McBride, & Trimble, 2017).

These ideas are directly applicable to distributed systems, as the notion of NP-hard problems naturally occurs in many places. If our data structures and algorithms can be modified so that the decision trees can avoid extensive evaluations, then the solution is returned faster. It would also mean that the memory footprint would decrease, allowing for more efficient usage of hardware resources. After these preprocessing optimizations have evaluated, then distributed systems enable concurrent execution of the remaining branches across a High-Performance Computing Service (HPCS) (Thiele et al., 2014).

# Conclusion

Distir