Trading Locality for Popularity

Michail Michailidis

Brown University   
mmichail@cs.brown.edu

Ugur Cetintemel

Brown University   
ugur@cs.brown.edu

Tim Kraska

Brown University kraskat@cs.brown.edu

Stan Zdonik

Brown University   
sbz@cs.brown.edu

**1. ABSTRACT**

Interactive Data Exploration (IDE)[1] is a multi-step non-linear process that typically involves unstructured datasets and imprecise end-goals. Most discovery-oriented applications from fields like scientific computing, financial analysis, evidence-based medicine and genomics, fall in the category of IDE. The need for more efficient IDE is becoming more apparent as data sets increase in size and complexity.

Since big datasets don’t fit in the main memory, caching is being used. Traditionally caching can be done on disk page level but there is a need for caching on tile level, which is a visual element. Scientific data is usually spanning among multiple dimensions. Since IDE is not easy to predict, cache eviction policies like LRU are not enough and there is a need for more sophisticated ones that are based on likelihood of future actions and machine learning models (learners).

In this work, we propose a middleware between the visualization and the storage system that sits on the client side. The main goal of the middleware is to minimize the communication and data transmission between server and client. This is possible with its cache management system that caches the tiles. Given the assumption that the visualization provides basic user actions of pan, zoom and jump, the main novelty of the cache is that each tile is broken into fragments that are not overlapping and can be used to recompose the original tile. Essentially fragments are lower resolution tiles and zoom levels can be defined according to how many fragments a tile in the viewport contains. When a tile is needed by the visualization layer, only the fragments of that tile that are not already in the cache are requested from the server, according to the current zoom level, resulting in less data transmission. Having stratified/fragmented tiles, increases the number of tiles that can be stored in cache, since there will be partially stored ones. Apart from the impact of partially stored tiles, another important topic is the tradeoff between popularity and locality using a formula that combines those two. Some workloads could be very locality-oriented, while others are not. We will see that different ways to combine locality and non-locality give different results in each of the workloads. Finally we will study the impact of memory size, while comparing full resolution tiles and variable resolution tiles.

**2. INTRODUCTION**

IDE applications involve complex large data sets[5], sometimes in high dimensions and lacking structure. The exploration process is multi-step, non-linear and the end goals are not well defined. Taking into account that datasets are increasingly becoming bigger, there is a need for more optimization approaches, as smaller fraction of data can fit in main memory. The good news is that user steps are not completely unrelated and there is causal relation between them. Also user actions usually have a tendency to correlate with the underlying dataset qualities.

Examples of current systems, Seer[2] and Scout[3], both operate in a multidimensional space. Seer predictions are based on experts that are summarizing movements of a group of users. e.g users that are interested in specific parts of the data in contrast to others. So when a user explores the data, the system tries to understand which expert is more likely to resemble this user. In contrast, Scout predictions are taking into account the internal structure of data. For example when we have neurons or streets, the system can predict that the users will pan towards main streets or main neurons, so it will prefetch these parts of data with higher probability than other nearby areas.

When prefetching and caching is needed, the data should be divided in smaller pieces, that someone can keep track and reason about their importance and likelihood in the future. This is the main reason that data is usually divided in tiles, which are rectangular, uniform, non-overlapping and possibly multidimensional. Each of the tiles can be imagined as a set of values for each of the dimensions of the dataset and encloses part of the data.

The prediction system of this work assigns an importance metric on the tiles that is calculated based on some static popularity metric that could be derived from how many times a user visited a specific tile, and the distance from current user’s position. It is worth noting that the metrics are defined on a tile basis and not on fragment basis.

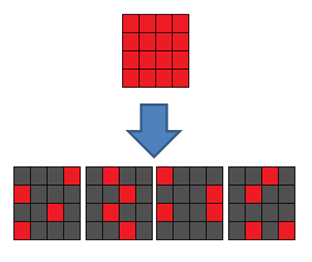
In this work, we assume that the actions available for the user are pan, zoom and jump. In order to support zoom, there is a need of tiles of different resolutions. We divide the tiles in smaller non-overlapping parts called fragments. Fragments have almost the same amount of data. When we have all the fragments of a tile stored, we have a tile in its full resolution or in highest zoom level. Whenever the user zooms in one level, she requests more data from this specific tile or in other words, one more fragment of it.

# 3. TILES AND FRAGMENTS

In IDE applications it is very common to have the data broken into tiles. There are many reasons for that. Tiles are uniform, rectangular and there are analogs of tiles in higher dimensions if needed. Having data as tiles, makes for easier database querying and easier way to track user’s moves. In addition a prediction system can be built with tile as a building stone. In such a system, probabilities or statistic of some sort can be kept for each of the tiles, something otherwise impossible. Division of the data in tiles also allows the implementation of a cache manager. When a tile is visited by the user, this tile is being stored in the cache, so if the user revisits the tile, there won’t be a need to query that tile from the database.

Data tiles were inspired from map tiles found in systems such as Google Maps[7].They can be imagined in different ways. They can be thought as images with each of the pixels representing a specific scientific measurement (e.g temperature, humidity etc). This measurement can be then converted to a range from 0 to 255 which corresponds to a color range of a single channel. That conversion allows rendering raw data of a tile into a monochrome image. Another tile representation is just a space boundary with points of interest in it. Both views of a tile are similar and can be converted to each other. In any case a tile is a (possibly ordered) set of values with specific coordinates for each of the values and can be stored as an image file, in an array, spatial or relational database, depending on the density and the geometric qualities of the values.

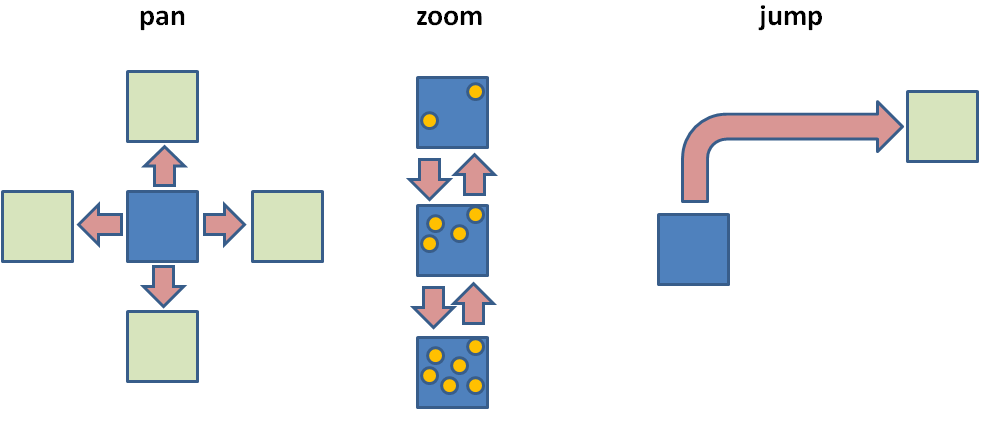
It is possible that a tile contains too much data that cannot be visualized properly or we want the user to be able to interrupt a tile query but still get partial information. There are many ways of data reduction[4][7] like binned or online aggregation that do sampling impacting the visualization in various ways, that we are not going to explore. In this work we assume the need for tiles of lower resolution. We introduce fragments that are similarly sized subsets of pixels or data points of a tile. Fragments are not overlapping and are the result of some specific uniform sampling technique for all the tiles. The number of fragments per tile corresponds to the number of zoom levels that the application provides. Unlike imMens[7] that is doing data reduction to increase frames per second of rendering, our motivation for breaking tiles in fragments, is to decrease miss ratio and user latency by caching partial tiles that take less space. An additional consequence of using partial tiles is the decrease in data transferred between client and server. *Figure 1* shows how fragments are defined using random sampling. The non-overlapping division could be in pixel level if data is in an image format or subsets of points



**Figure 1: A full tile divided in four fragments**

# 4. USER ACTIONS

In this work we assume that the application provides a viewport, which is what the user can see and three actions: pan, zoom and jump (see *Fig. 2*).



**Figure 2: User actions (pan, zoom and jump)**

**Pan**

The user can go in four directions: up, down, left and right. The viewport moves accordingly in those directions and a diagonal move is a combination of two primitive moves.

**Zoom**

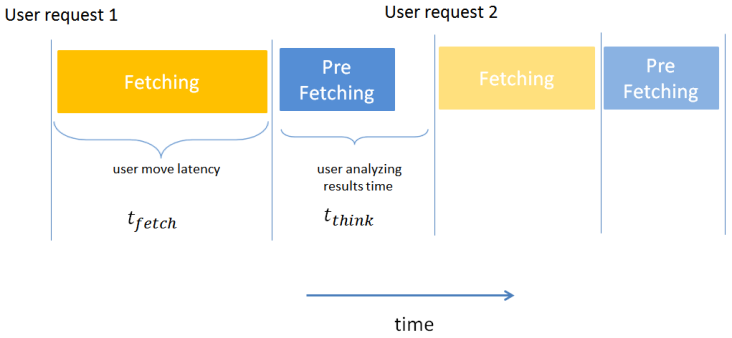
Zoom is possible due to fragmentation of a tile. If a tile has 16 fragments then 16 levels of zoom are available. Tile in zoom level 1 has the least information (one fragment) and a tile in zoom level 16 is full. We have a zoom-jump when a user goes from one zoom level to another in a single move. In all cases the viewport doesn’t move.

**Jump**

Jump is the action, that teleports the viewport from any place to any other that the user wishes to visit. The Google Map analog would be, when a user types some text and searches for a place to start her exploration.

# 5. USER THINK TIME AND LATENCY

Each time a user does a pan, zoom or jump a request is sent to the server. The request contains only the fragments that are needed from that action and are not available in the cache. The time that is needed for those fragments to be returned from the server is called fetch time (tfetch) and is the latency that the user experiences in an application. After the fragments have been returned to the client, the time that follows till the next user action is called prefetch window and is the think time (tthink). During that time the user processes what she sees, till she finds something interesting and takes a new action. Prefetch window is also the timeslot that allows us to predict according to what the user is up to and what we could prefetch in advance. If we do a good job the next fetch time will be shorter because some of the needed fragments will be in the cache already. When something that we need is already in the cache we have a cache hit and when something is not there we have a cache miss. Prefetching when done effectively causes more cache hits that subsequently reduce latency or fetch time. *Figure 3* summarizes latency and think time.



**Figure 3: Fetch time and think time used for prefetching**

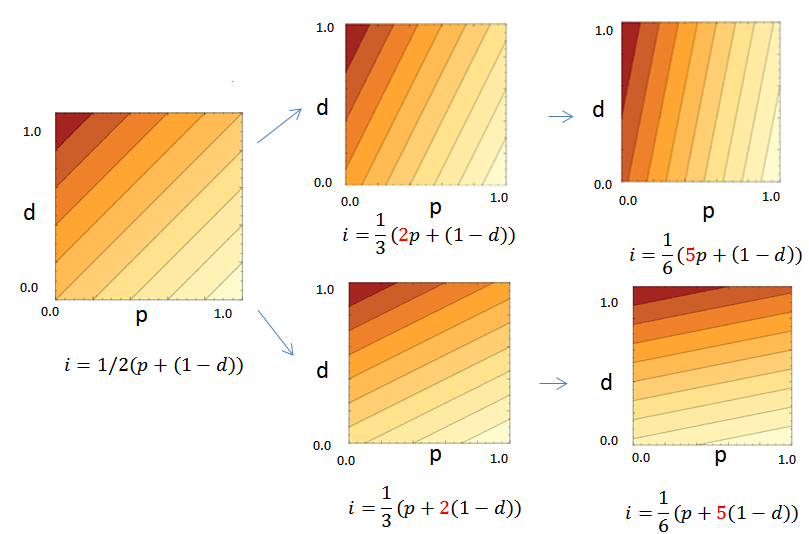
# 6. METRICS

An effective caching and eviction mechanism needs a metric to make tile comparison possible. A simple one is Euclidian distance measured from current position to the tile in hand. Distance is a very good metric if a workload has high locality. In such a workload the user would make small steps or in other words would only pan. With distance as a metric the closer tiles are prioritized and prefetched ahead of the farther ones. If there is always enough think time for the four surrounding tiles to be prefetched, there won’t be any cache misses and latency is zero, with the exception of the first tile of user movement.

While distance could be a good metric for locality-oriented workloads, that’s not true for most workloads. In case there are jumps, there should be another metric that is not dependent on distance we call popularity. Popularity is the metric of how often a tile was visited through pan or jump.

Given a workload or a user study on a specific dataset, popularity is a value between 0 and 1 that can be assigned to each of the tiles. This (normalized) value can be derived from the number of times a user has visited this tile divided by the biggest number of visits in the whole dataset. Similarly Euclidean distance of a tile can be normalized by dividing with the length of the diagonal of the rectangular dataset. This length is the longest distance any tile can have from any other tile. Since the longer the distance of a tile from the current position, the worse the tile is, we subtract it from 1. Now having two metrics, suggests the need of a combination formula.

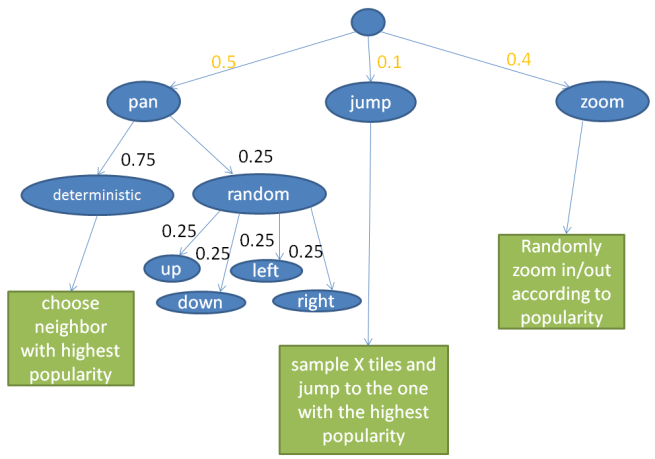
There are many different ways to combine popularity and distance metrics. The combination formula should be the simplest and should allow weights in front of the popularity and distance metrics. One formula that has those characteristics is a linear combination of the two after normalization: *importance = (a\*(1-distance) + b\*popularity) / (a+b)*. When either *a* or *b* is equal to zero then *importance* depends on one of the metrics only and when *a* equals *b* then importance equally depends on both distance and popularity. *Figure 4* shows how changing one of the weights affects the slope of importance plane.



**Figure 4: User actions available (pan, zoom and jump)**

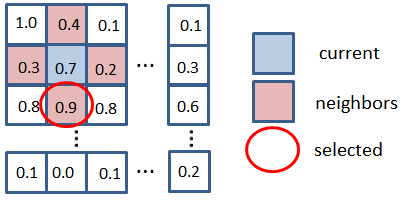
# WORKLOAD GENERATOR

Workload generation is very simple. As *Figure 5* shows there are three main branches in the tree of workload generator. Each of the branches corresponds to a user action and has a probability that determines how often this action will be taken. All three probabilities add to 1 and only the top three numbers (with yellow) differ from workload generator to workload generator.



**Figure 5: Workload Generator**

The first branch is pan. Pan has two sub-branches: one that is deterministic and the other one that is random. Deterministic happens 3 out of the 4 times that a Pan happens. In this case, user moves to neighboring tile with the highest popularity as shown in *Figure 6*. Otherwise, one of the four neighboring is being selected at random. Randomness helps preventing user from being stuck in local extrema.



**Figure 6: Deterministic Pan**

The second branch is jump. Jump is assumed that happens more frequently on tiles that are more popular. If we would order the tiles based on popularity and pick the most popular, the user would land on the same tile. Instead a random sample of 10% is taken from the database and then the most popular tile is chosen. That way the simulated user tends to land on somewhat popular tiles.

Finally, the third branch is zoom but is a group of two actions: zoom in and zoom out. Given current tile’s popularity, the workload generator throws a coin that has head probability equal to popularity. If it is head it means that it will zoom-in and if it is not it will zoom-out. This setup promotes higher resolution levels for popular tiles and lower resolution levels for unpopular tiles.

# TILE CACHING AND EVICTION

In any cache management system caching is really important. When requested data already exists in the cache, the system can fetch it from there instead of the server, reducing latency. There are many different caching policies depending on the application. Usually they are variations of Least Recently Used policy (LRU). In case of LRU it is assumed that data accessed recently is going to be needed again. So if recent data is kept in the cache and older is evicted, the result is less cache misses and more cache hits.

For tile caching more sophisticated policies are needed. Usually those policies order tiles not in an LRU fashion but combined with a predictor, tiles are ordered based on a likelihood or probability metric[2][3]. The more likely tiles are kept in the cache and the less important are evicted. Likelihood is defined in terms of the probability that a user will need those tiles in the near future. In all cases a metric is used that defines the “hotness” of cache. Hotness can be time based, likelihood based or some combination of those, but the desired result is increasing cache “hotness” through caching of the valuable data and eviction of the less valuable.

As it was discussed in section 6, importance is used in this work, as a metric to order tiles. The tiles with highest importance are kept in the cache and the ones with lower importance are evicted. Importance is defined as a linear combination of distance and popularity, with the exception of the tiles in the current position of the viewport. Those tiles have infinite importance and are protected from eviction.

Systems that go one step further also incorporate prefetching. They use a predictor that was trained offline with workloads or user studies and are fetching parts of the data that the user is likely to find useful in the near future. In this work the prefetching mechanism, as caching mechanism, is using importance to decide what tiles are the best ones to prefetch. The cache manager goes through a lot of procedures that are going to be described below.

**User Move:** Each time the user moves, a fetch is triggered and importance of tiles in the cache is being updated because their distance from current user position has changed. That also implies that the order of the tiles has changed. In other words the tile that is on top is the one with the least importance and the one that is last is the current position tile. For simplicity we assume that the viewport only contains one tile.

**Fetch:** Fetch consists of a query to the database server, requesting the current tile, containing the number of fragments in accordance with current zoom level. If cache has the tile but not as many fragments as needed, only the ones that are missing are fetched. The tile that was fetched will be placed in the cache, with infinite importance so it won’t be evicted by subsequent tiles coming from prefetch. In case cache doesn’t have enough space eviction will happen.

**Prefetch:** After fetch has finished, prefetch starts. A list of all tiles ordered by importance at descending order is prepared. Each of the tiles about to be prefetched is checked against cache. If it doesn’t exist in the cache at all, it is being prefetched with as many fragments as the current zoom level suggests. If it is partially contained in the cache, then only the missing fragments are requested. Finally if the tile about to be requested is contained at the cache and needed number of fragments is stored, it is skipped from prefetching. One more case that an about to be prefetched tile, will be skipped is, if it has less importance than the tile with the least importance currently in cache. At the point there is not any think time left or all the tiles about to be prefetched have either worse importance than the worst stored one, or are already stored in the cache, prefetching stops. In this work the assumption is that think time is plenty, so the only constraint is cache memory size and what tiles are already in the cache.

**Eviction:** Eviction happens when memory is full or doesn’t have enough space for the new tile to be stored. Memory size is counted in terms of fragments. So assuming that a tile can have up to 16 fragments, and memory size is 1024, memory can hold up to 64 full tiles or 1024 fragments. In case of full tiles mode, cache is always full, but in case of partial tiles mode, cache can have available space for a small number (less than 16) of fragments.

Eviction algorithm has to be greedy, since it doesn’t know what the next tile to be prefetched is. The only information it has is that the next tile has equal or worse importance than the current one. When a tile is about to be stored and has x number of fragments, x number of fragments have to be evicted. Eviction happens in a tile basis so, tiles are being removed from the tile till available space is at least x fragments. Obviously this could cause more tiles than one to be evicted, so that enough space is available for the new tile. In rare cases after removing a sequence of tiles, the worst tile in the cache is better than the one to be stored. In that case storage of the new tile doesn’t happen.

# EXPERIMENT SETUP

**Database:** MySql is used as a back end. Database is about 1Gbyte and has 625 tiles (25 x 25) tiles or 10000 fragments. Each tile is 256x256 pixels and has 16 fragments. Fragments are stored as rows of a table with schema (y, x, fragment\_number, data). Tile coordinates are given by y and x. A tile is comprised of 16 rows. Data is of type text and contains color intensity information of the pixels in the specific fragment.

**Cache Misses:** Cache misses are counted in a fragment basis. That means if a whole tile is missing and the max number of fragments a tile can hold is 16, then there will be 16 misses.

**Think time:** Think time is not constrained.

**Cache Size:**  Cache size is defined in terms of fragments. So size of 256 is exactly 16 full tiles or 256 fragments that can be distributed in up to 256 one fragment tiles. The first series of experiments is run with cache of size 256 and the second series with 128,256,512,1024,2048,4096 and 8192.

**Workloads:** In this experimental setup six different workload generators were used: pan heavy, pan-only, jump heavy, jump-only, zoom heavy and a mixed one. After the first series of experiments best metric is selected for each of the workloads and then cache size is varied for the second series of experiments.

**Metrics:** This general form of mixing formula is used: *(a\*p+b\*(1-d))/(a+b)*. Weights *a* and *b*, take values from 0 to 5 but are never equal. In the results a label of *p, 3\*(1-d)* is a shorthand for *(p+3\*(1-d))/4.*

**Goal:** The goal of the experiment is to explore the usefulness of fragments compared to tiles. Another goal is to explore ways to make prefetch adaptable to different locality workloads by using different mixing metrics. Cache size impact to performance is also explored.

# RESULTS

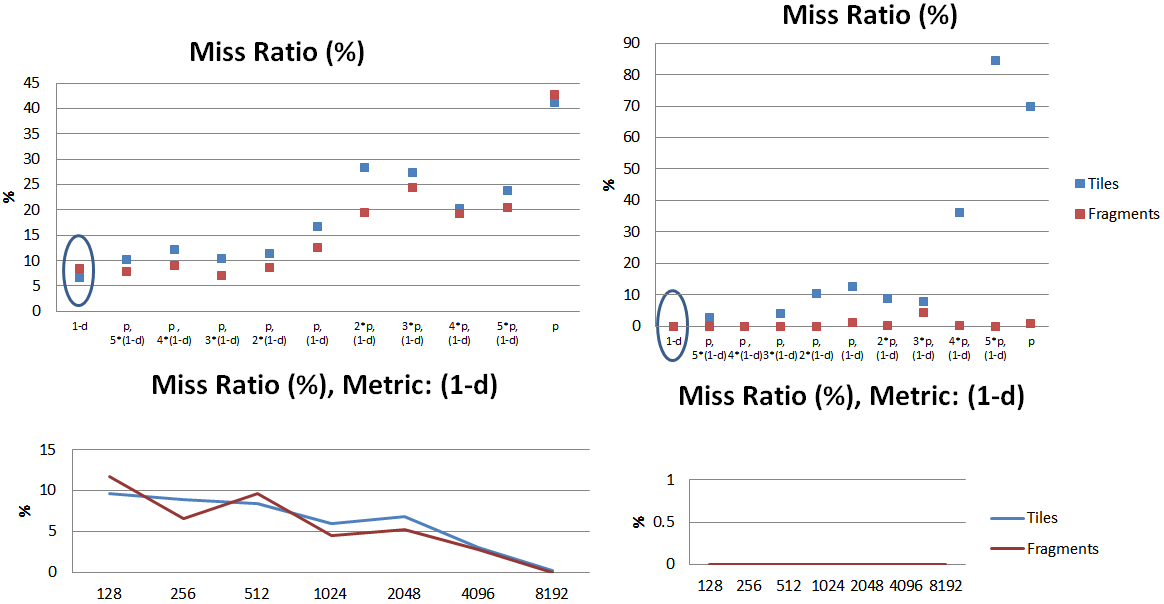
Six workload generators were used and are described below. Sometimes the terms *workload generators* and *workloads* are used interchangeably.

**Pan heavy and Pan only:**

Pan heavy (Figure 7a) workloads have 80% pan, 10% jump and 10% zoom moves, while pan-only workloads have 100% pan moves. In this experiment we see that metric (1-d) is the most effective for both of them, which is quite expected. Someone would probably expect a less local metric for Pan-heavy workloads, that’s why any of the first three metrics is a good choice. Given these metrics we vary the cache size and we see that there is no clear winner between fragments and tiles for either of the workloads.

In Pan-only (Figure 7b) we see that the performance is the same. We have to keep in mind that in pan-only, no zoom action is taken so tiles have the lowest possible resolution of 1. The performance is the same because for tiles there is need for 4\*16=64 fragments cache size for 0 misses and for fragments this is just 4\*1=4 fragments. In either case the cache size is way bigger than what this very local workload needs.

For tiles in pan-only, when metric is *5\*p, (1-d)* there is a somewhat unexpected peak. It is somewhat unexpected because this workload is very local and a less local metric (*p*) behaves better than a more local one (*5\*p, (1-d)*). What actually happens is that *5\*p, (1-d)* encourages prefetching of quite popular tiles but also quite close to current position (not the closest). Pan chooses the neighboring tiles that were not cached. In contrast (*p*) prioritizes fetching of the most popular tiles and we know that most of the times pan will go to the most popular neighboring tile.



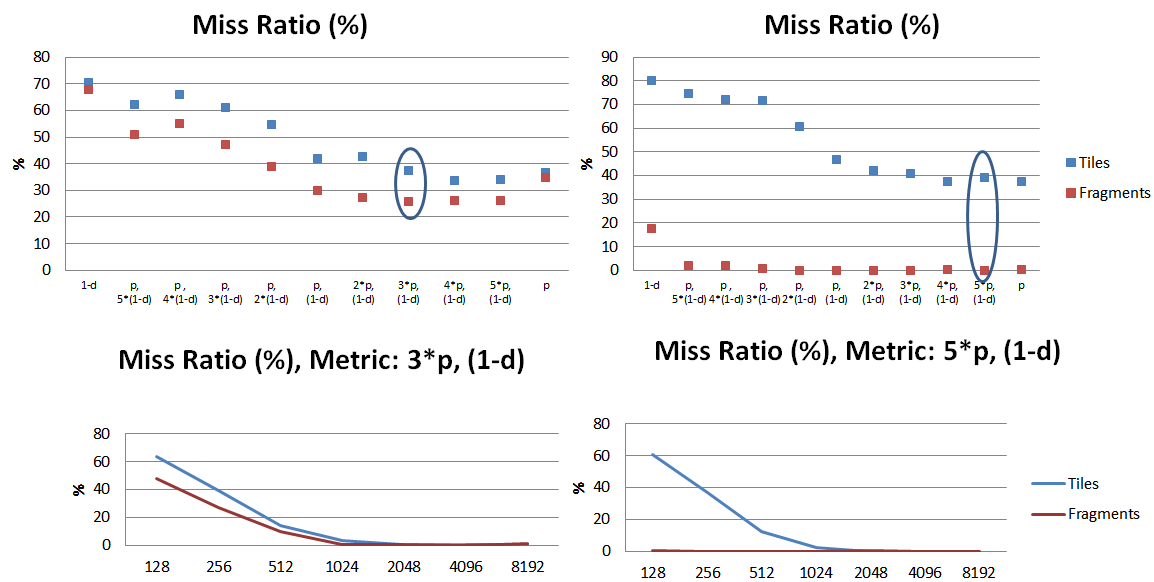
**Figure 7: a) pan heavy (pan 80%, zoom 10%, jump 10%)**

**b) pan only (pan 100%, zoom 0 %, jump 0%)**

**Jump heavy and Jump only:**

Jump heavy workloads have 80% jump, 10% pan and 10% zoom moves, while jump-only workloads have 100% jump moves. In *Figure 8* we see that less local metrics are the most effective. For jump heavy workloads *3\*p,(1-d)* is the most effective giving the lowest miss ratio, while for jump-only, *5\*p,(1-d)* is the most effective however very close to *p.* It seems that for jump heavy workloads fragments can give a performance boost of up to 50% for smaller cache sizes.

For jump-only workloads we see a more than 370x boost but that is because in fragments mode, tiles have the lowest possible resolution of 1 fragment. In other words, with tiles that have only one fragment, it is like having cache memory size 16 times bigger. This experiment is only useful for estimating a good upper bound of performance increase for non-local workloads.



**Figure 8: a) jump heavy (pan 10%, zoom 10%, jump 80%) b) jump only (pan 0%, zoom 0 %, jump 100%)**

**Zoom heavy:**

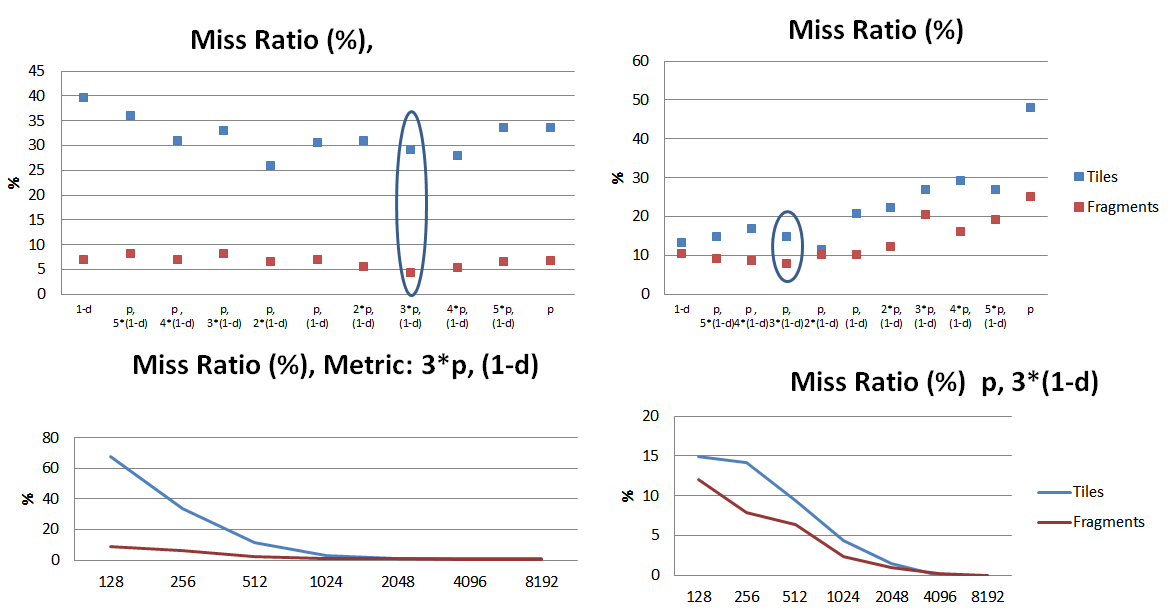
Zoom heavy workloads have 80% zoom, 10% pan and 10% jump moves. In those workloads (see *Fig 9a*) we see that most metrics behave about the same, but the best is *3\*p, (1-d)*, which is a bit on the non-local side of metrics. That is expected as zoom in is more likely to happen on tiles that are more popular. The impact of metrics is minimal because zoom happens in the current viewport and as time goes by, tiles accumulate more fragments in cache, usually though the tiles visited. The main difference in performance between tiles and fragments, is due to full tiles taking much more space than partial ones.

**Mixed:**

Mixed workload has 50% pan, 40% zoom and 10% jump. We try to simplistically model movement for a user of Google maps, where a user jumps once and then spends most time panning and zooming till he jumps again. Jump happens when the user searches for a new area to explore. As *Figure 9b* shows, since this workload has a lot of locality, the best metric is *p, 3\*(1-d)*. Given that metric, we can have up to 80% boost having fragments enabled.

**Overall:**

With the exception of pan and pan-only**,** fragments have an advantage over full tiles especially in zoom heavy workloads. In all those workloads smaller cache sizes make fragments impact in performance bigger.



**Figure 9: a) zoom heavy (pan 10%, zoom 80%, jump 10%) b) mixed (pan 50%, zoom 40%, jump 10%)**

# FUTURE WORK

Interactive Data Exploration hides a lot of challenges and there is still a lot of work to be done. This work’s results can be used as a motivation for other more sophisticated methods to surface. As synthetic workloads and datasets were used a lot insights could arise from real ones. It is very interesting to see if datasets stored in an array database like SciDB[6] could benefit from this work. The impact of think time could give even more room for fragments to have even more significant advantage over tiles as the user can interrupt queries and still get partial results. The prediction was based on a very short-sighted popularity grid of values that can be derived from a user study or workload. It would be interesting to explore the combination of mixing formulas with more sophisticated prediction systems. In current work, viewport has size of one tile. It is interesting to see how all these user actions and the prediction system would have to be transformed to accommodate overlapping pans, grouped zooms and jumps. Finally there is need for sensitive analysis. Results of one workload could be used against a slightly different workload to explore robustness.

# REFERENCES

[1] Query Steering for Interactive Data Exploration - Ugur Cetintemel, Mitch Cherniack, Justin DeBrabant, Yanlei Diao, Kyriaki Dimitriadou, Alex Kalinin, Olga Papaemmanouil, Stan Zdonik - CIDR’2013

[2] Seer: Predictive Prefetching and Caching in Large Scientific and Analytic Datasets - J. DeBrabant, U. Cetintemel, S. Zdonik.

[3] **SCOUT: prefetching for latent structure following queries – Farhan Tauheed, Thomas Heinis, Felix Schurmann, Henry Markam, Anastasia Alimaki, Proceedings of VLDB Endowment,** Volume 5 Issue 11, July 2012   
Pages 1531-1542

[4] ScalaR: Dynamic Reduction of Query Result Sets for Interactive Visualizaton – L. Battle, R. Chang, M. Stonebraker

# [5] **EarthDB: scalable analysis of MODIS data using SciDB – Gary Planthaber, Michael Stonebraker, James Frew**

[6] Overview of SciDB: Large Scale Array Storage, Processing and Analysis – The SciDB Development Tea

[7] imMens: Real-time Visual Querying of Big Data – Zhicheng Liu, Biye Jiang and Jeffrey Heer, Eurographics Confernece on Visualization 2013, Volume 32, Number 3

[8] Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines