Analytics of Game Player Data using Datalog

Eric Butler (edbutler@cs), Aaron Bauer (awb@cs)

June 7, 2013

Introduction

Analyzing player data from our lab's (the Center for Game Science) educational games at a large scale is an important component of many ongoing research efforts. Several such projects [1, 4, 7, 5] involve computing complex features on data from as many as several thousand players. Currently, due to the complexity of the features we need to compute, such analysis is done ad hoc with hard-coded scripts or programs, in languages such as Python or Java. While it would be desirable to move this computation into a database system, such queries are cumbersome or even impossible to express in SQL. In this project, we explore the use of Datalog, in particular LogicBlox [2], to determine whether it can concisely express and efficiently compute the kinds of queries we need. We draw on previous work [7] for example queries, and write a Datalog$^{\text{LB}}$ (Datalog for LogicBlox) program to compute them. Originally, the queries were computed with a hand-coded system written in a mix of Python, Scala, and Answer-Set Prolog, the bulk of the system and computation being in Scala. The rest of the paper is as follows: we first discuss background and related work and then describe the Datalog program we created. We then define the queries we wish to answer and evaluate our system using those queries. Our evaluation has three components. First, we measure how the Datalog program performs as the data size increases, looking at a dataset with 10,000 players. We then compare this performance to that of our existing system. Lastly, we discuss qualitatively the ease of writing programs and expressing queries in both systems.

Background and Related Work

The game on which we will be computing queries is *Refraction*, an educational fractions game that involves splitting lasers into fractional amounts. The player interacts with a grid that contains laser sources, target spaceships, and asteroids, as shown in Figure 1. The goal is to satisfy the target spaceships (each one desires a specific fractional laser value) and avoid asteroids by placing pieces on the grid. Some pieces change the laser direction and others split the laser into two or three equal parts. To win, the player must correctly satisfy all targets at the same time.

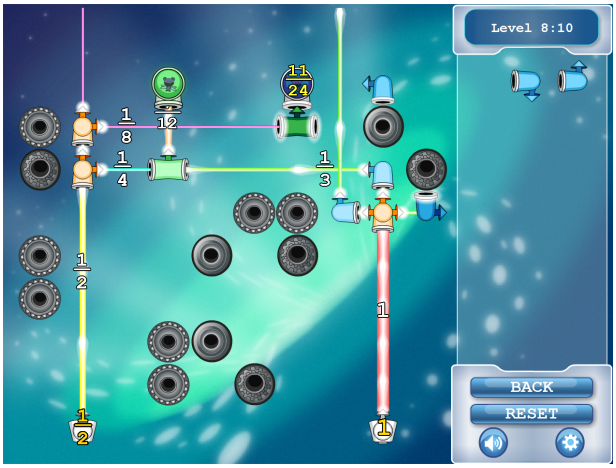

Figure 1: A level of *Refraction*.

There is a large body of previous work on game analytics. Much of this work focuses on systems for visual exploration and analysis, emphasizing aspects such as investigating player experience [3], understanding numerical data [6], or representing players' paths through the game [1, 10]. Our project aims to facilitate asking such sophisticated queries at scale, and we leave improving the accessibility and discoverability of our system as future work.

Our lab uses hard-coded programs for the vast majority of feature computation and analytics. There is a lack of information on the processes typical game studios in the industry use for analytics; however, publications such as those about Microsoft's in-house game data analytics group [3] and the analytics tool for Electronic Arts' game *Dead Space 2* [6] indicate

that most of the computation happens outside of the database system. While general purpose programming languages have the expressiveness for our queries, there are two obvious drawbacks. First, performance becomes a serious issue at scale. When the data needing to be processed no longer fits in main memory, programs to effectively process this data become quite complicated. Second, and rather more importantly, exploration of new features is a slow iterative process. Slightly adjusting a query requires modifying a large program, recompiling, and redoing much of the computation. Among their many uses, database systems are effective at overcoming precisely these two problems.

Although our player data starts in a SQL database, we do not use SQL for much of the analysis. This is primary because it is difficult to get the expressiveness we need within SQL; many of the features we look at require simulation of the game. For example, in *Refraction*, we care about counting the number of unique "game states" a particular player traversed during a playthrough. This requires simulating the game, which requires recursion to compute the flow of lasers throughout the board. Not only is this difficult to express in SQL, such games may even be outside of the complexity class expressible by standard SQL. For games such as *Light-Bot* (Coolio Niato 2008) and *SpaceChem* (Zachtronics Industries 2001) this simulation approaches a **P**-complete problem: they require simulating a player-designed machine for a bounded number of steps. Even with extensions to SQL that support such complexity, the language does not lend itself well to concisely expressing the large, complex set of rules.

The work of White et al. [11] has tried to simulate games in a database context, using a custom language to overcome the verbosity of SQL. They propose a declarative scripting language called Scalable Game Language (SGL) that compiles down to SQL. SGL requires designers to implement the game logic according to a "state-effect" design pattern such that the state of each game object is updated once at the end of each simulation loop by aggregating all the independent effects applied during a prior "query phase." It is this separation that allows SGL to provide an order of magnitude better performance. This system does not capture the expressiveness necessary to simulate *Refraction*, however, and is thus not suitable for our purposes.

Type	Ins.	Outs.	Description
Source		1	Emits lasers
Target	1 to 4		Must satisfy to win
Bender	1	1	Bends laser 1/4 turn
Splitter	1	2 or 3	Equally splits lasers
Combiner	2 or 3	1	Sums lasers
Blocker			Obstructs all lasers

Table 1: The different piece types of *Refraction*.

Logic programming is a more expressive language that makes representing deductive rules straightforward. In fact, our lab has had success using Answer-Set Prolog (ASP), a variant of Prolog for answer-set programming, to generate levels for *Refraction* [9, 8]. The answer-set program contains, in Prolog-like rules, the logic required to simulate *Refraction*. However, ASP is designed for model checking and satisfiability of small domains, and is not suitable for computation on the scale of thousands of players and millions of actions.

It is thus natural to try to use Datalog for our queries. If a Datalog program could concisely express the rules needed to simulate *Refraction*, then perhaps many of our queries could be computed entirely within a database system.

System Description

Data Schema and Description

The particular dataset we are using consists of player data from the version of the game on the Flash game portal Kongregate[1]. The full dataset contains approximately 15,000 players, 400,000 levels played (which we call *traces*), and 15,000,000 actions on those traces. We used a subset of three levels that contained 200,359 actions.

Listing 1 shows the schema for our extensional data in Datalog$^{\text{LB}}$ syntax. This data is broken down into two hierarchies: level data and player data. The level

[1] `http://www.kongregate.com/`

data describe, for each *level* in the game, which *pieces* are available in each level and their attributes. Pieces are grouped into different types, which are described in Table 1.

Player data is broken into several tables. At the lowest level are *actions*, which are the individual actions performed in levels, such as picking up a piece. Above this are *traces*, which group actions into an instance of a particular player playing a particular level. Above this are *sessions*, which group traces into play sessions, and at the top are *players*.

```
Level(l), hasLevelId(l:i) -> uint[32](i).

BoardLocation(bl), hasLocation(bl:i) -> uint[8](i).
bl:xOf[bl] = x -> BoardLocation(bl), uint[8](x).
bl:yOf[bl] = y -> BoardLocation(bl), uint[8](y).

Direction(d), hasDirectionName(d:s) -> string(s).
d:axisOf[d] = i -> Direction(d), uint[8](i).
d:signOf[d] = i -> Direction(d), int[8](i).

Fraction(f), hasFractionRepr(f:s) -> string(s).
f:numOf[f] = n -> Fraction(f), uint[8](n).
f:denOf[f] = d -> Fraction(f), uint[8](d).

Piece(p) -> .
piece:levelOf[p] = l -> Piece(p), Level(l).
piece:tagOf[p] = tag -> Piece(p), string(tag).
piece:idOf[p] = id -> Piece(p), uint[8](id).

piece:input(p, d) -> Piece(p), Direction(d).
piece:output(p, d) -> Piece(p), Direction(d).
piece:valueOf[p] = f -> Piece(p), Fraction(f).

SourcePiece(p) -> Piece(p).
TargetPiece(p) -> Piece(p).
BlockerPiece(p) -> Piece(p).
BenderPiece(p) -> Piece(p).
SplitterPiece(p) -> Piece(p).
CombinerPiece(p) -> Piece(p).

Player(p), hasPlayerId(p:s) -> string(s).

Session(s), hasSessionId(s:i) -> string(i).
session:playerOf[s] = p -> Session(s), Player(p).

Trace(t), hasTraceId(t:s) -> string(s).
trace:sessionOf[t] = s -> Trace(t), Session(s).
trace:levelOf[t] = l -> Trace(t), Level(l).

Action(a), hasActionId(a:s) -> uint[32](s).
action:traceOf[a] = t -> Action(a), Trace(t).
action:seqIndexOf[a] = i -> Action(a), uint[16](i).
action:typeOf[a] = s -> Action(a), string(s).
action:timeMillisOf[a] = t -> Action(a), uint[32](t).
action:pieceIdOf[a] = pid -> Action(a), uint[8](pid).
action:locOf[a] = bl -> Action(a), BoardLocation(bl).

StartAction(a) -> Action(a).
PickupAction(a) -> Action(a).
DropInBinAction(a) -> Action(a).
DropOnBoardAction(a) -> Action(a).
ResetAction(a) -> Action(a).
```

Listing 1: The schema for our EDB. It is primarily broken in two categories: level data and player data.

Queries

Before defining the three queries we ran, we discuss the common elements between them: simulating the game rules of *Refraction*.

Recursively Computing States

For all of our queries, an intentional table for *states* must be computed. This consists of several components, many recursively defined. First, player actions must be recursively simulated to compute a `pieceAt(action, piece, boardloc)` relation that describes where pieces are on the board after each action. Piece locations define a *game state*, so we then associate each action with the game state it engenders. Once we have states, we compute several features on them. The most complex part of this computation is constructing the *laser graph* for each state. The laser graph describes the fraction-valued lasers currently on the board. Starting with source pieces, the lasers must be pushed around the board to the remaining pieces, which requires multiple recursions. Construction of this laser graph a primary reason hard-coded programs have been required in the past. We include part of the Datalog$^{\text{LB}}$ code for this computation in Listing 2.

```
outEdge(s, p, dir) <-
    SourcePiece(p),
    isPieceOfState(s, p),
    piece:output(p, dir).
movingLaser(s, p, bl, dir) <-
    outEdge(s, p, dir),
    pieceAt[s, p] = bl.
laserReaches(s, p, bl, dir) <-
    movingLaser(s, p, blold, dir),
```

```
    nextCell[blold, dir] = bl.
movingLaser(s, p, bl, dir) <-
    laserReaches(s, p, bl, dir),
    !isCellOccupied(s, bl).
dirEdge(s, pout, pin, dir) <-
    piece:input(pin, opdir),
    oppositeOf[dir] = opdir,
    laserReaches(s, pout, bl, dir),
    pieceAt[s, pin] = bl.
edge(s, pout, pin) <-
    dirEdge(s, pout, pin, _).
outEdge(s, p, dir) <-
    edge(s, _, p),
    piece:output(p, dir).
```

Listing 2: The recursively-defined edge relation, part of the code that computes the "laser graph" for each game state s. The output is `edge(state, piece_out, piece_in)`, which describes which pieces' outputs and inputs are connected on the game board. The system computes this by, starting with source pieces, recursively pushing lasers around the board until the reach the input of another piece.

With this supporting structure in place, we were able to compute three sophisticated queries of interest.

The Three Queries

1. For each trace, what was the ratio of unique states to total states? We have previously identified this metric as an important feature for player prediction. To compute it, we count number of distinct states in a given trace and divide by the total number of states for that trace.

However, we want to look at a particular representation of states, a feature we call the *fringe lasers*. The *fringe lasers* of a given state are the set of lasers that are not being correctly used as the input of any piece on the game board. For each state, we define a relation `fringeLaser(state, piece, direction, fraction)` that has an entry for every fringe laser value on the board. Note that multiple fringe lasers with the same value may exist, so the piece and direction are included in the relation to allow for the equivalent of bag semantics.

When selecting the count of distinct states, we must not double count states that have the same bag of fringe laser values. To address this we wrote an equivalence relation using the `fringeLaser` predicate and some additional rules to count the number of equivalence classes. This code is shown in Listing 3.

```
// starting with an equivalence relation,
// we must compute equivalence classes
equalStates(s1, s2) -> State(s1), State(s2).

// do this by choosing a canonical
// representative of each class via
// an arbitrary ordering
better(s1, s2) <- s1 < s2, equalStates(s1, s2).
best(s) <- State(s), !better(s, _).
canonicalStateOf[s1] = s2 <- better(s1, s2), best(s2).

canonicalStateOfTrace(t, canonicalStateOf[s]) <-
    action:stateOf[a] = s, action:traceOf[a] = t.

numUniqueStates[t] = cnt <-
    agg<<cnt = count()>> canonicalStateOfTrace(t, _).
numActions[t] = cnt <-
    agg<<cnt = count()>> action:traceOf[_] = t.
proportionUnique[t] =
       uint32:float32:convert[u] /
       uint32:float32:convert[a] <-
    numUniqueStates[t] = u, numActions[t] = a.
```

Listing 3: The definition of `proportionUnique(t, p)`, the final part of the query for proportion unique states. Stating from an equivalence relation, we must count the number of equivalence classes in each trace. We accomplished this by imposing an arbitrary total ordering on states and computing a "canonical" state for an equivalence class to be the "largest" state in that class. We can then count the number of distinct canonical states to compute proportion unique.

We will call this query the *proportion unique* query.

2 and 3. How often does the player make mathematical mistakes? One major goal of the game is to estimate whether students are improving at math. One subgoal is to estimate when students make mathematical mistakes, which we can do by computing the erroneous game states a player visits. The following queries are concerned with two possible definitions of erroneous states.

In the first query, we define an erroneous state as one where there exists at least one target with an

input laser of the incorrect value. We will call this the *wrong sinks* query.

For the second query, we define an erroneous state as one in which there exists a target such that that target cannot be satisfied with the remaining fringe lasers and splitter pieces. For example, suppose the targets are 1/9 and 1/6 and the source laser is 1/1. If the player splits the laser in 2 halves, then there are no splitting operations to go from 1/2 to 1/9, so the state is erroneous. We will call this the *bad splitters* query.

Evaluation and Discussion

We have implemented a functional analytics system in Datalog$^{\text{LB}}$ to compute the answers to the proposed queries. We have verified our systems accuracy by comparing the results of our Datalog queries to the results generated by the previously existing system. To evaluate our new system, we performed two analyses.

Experimental Performance Results

The first was a quantitative analysis of the performance of our datalog queries as a function of the size of the input. We computed each of the three queries for three different Refraction levels (50, 53, and 54) with varying sizes of input. Specifically, we varied the number of players whose data was imported from 1 player (as a baseline) up to 10,000 players. We sampled more densely (increments of 50) for player counts under 500, and more sparely (increments of 100, and then 500) above that. Hence, each data point corresponds to the running time for a unique combination of query, level, and input size.

To obtain reliable results, we ran five trials for each data point, discarded the the highest and lowest times, and averaged the remaining three times. Each trial measured the following steps: (1) generating appropriate csv files from the raw player data, (2) creating a new database and importing everything fresh (schema, rules, etc.), (3) importing the player data into the database, and (4) outputing the query results.

Using these results, we compared the performance to that of the original system. Charts showing the results of this analysis can be found below. For all queries, it's clear that our Datalog system performs far better at scale, with the exception of *proportion unique*. In this case, the query has quadratic performance, but for the amounts of data typically involved in an experiment (2,000 - 8,000 players), Datalog still outperforms the original system. Furthermore, we expect it would be possible to optimize the *proportion unique* query substantially. It is true, however, that additional development effort could be expended optimizing the original system to the point where it would outperform Datalog. Hence, the real advantage becomes apparent when considering performance versus development effort. The original system was the subject of months of work, whereas our Datalog system was built over the course of a few weeks, so in terms of the ratio of performance to development effort, the Datalog system is unambiguously superior.

Qualitative Discussion

As our second analysis, we qualitatively discuss the ease of use of the two systems. Our primary interest is to answer whether the use of Datalog systems would help with our research. How much better or worse is it to use Datalog compared to hand-coded imperative programs?

Besides performance, the other major benefit we hope to get from a database system is the ability to quickly explore and construct queries. Indeed, as noted above, it took us a significantly shorter amount of time to construct this system than the original. To gather some anecdotal evidence from another user, we asked colleague from our lab who has been directly involved in several of the studies of player data to explore Refraction data using our system. He has been using the original system for over a year to explore and analyze game data, but had little prior experience with logic programming.

After a 30-minute crash course in logic programming and the organization of our system, our colleague began experimenting with queries. After about 60 minutes of exploration and refinement, he

had successfully computed a feature of equal complexity to those he had been creating in code. He wrote a query that found the proportion of actions where a player put down a piece such that it blocked a laser, and did not use the laser (i.e., had no input on the side the laser hit). He built up this query iteratively, largely by copying and modifying existing predicates. While this is only anecdotal evidence, the speed and relative ease with which our colleague went from no familiarity with Datalog to computing a sophisticated query suggests our system could be effective in facilitating exploration of player data.

Conclusion

The queries our research requires are at a level of complexity that make SQL unattractive, so we have traditionally resorted to hard-coded programs. The datalog system has several obvious advantages over these systems. Obviously, as a DBMS, it performs well at scale and removes the need from the user of figuring out how to process data that does not fit into memory. Furthermore, it's declarative and concise syntax of logical rules allows for concise expression of *Refraction*'s game logic and rapid exploration of new queries.

There are some drawbacks. First, as Datalog is used very infrequently in industry compared to SQL, the tools and resources surrounding it are much weaker, and require some patience from the user. In fact, we even discovered a code generation bug in LogicBlox while developing our system. More importantly, however, the typical software engineer will be find some of the constructs required by logic programming quite challenging. In our example query *proportion unique*, the last step was to count equivalence classes. In imperative code this is straightforward: write an equals function and use a hash map. The logic programming solution, while concise, took a significant amount of time for us to think through. We expect that this will change if logic programming becomes more mainstream, but for now, the number of potential users is small.

However, we feel that for personal use, LogicBlox is an effective system for the kind of research the Center for Game Science does. We would recommend using such a system for future research endeavors.

References

[1] E. Andersen, Y.-E. Liu, E. Apter, F. Boucher-Genesse, and Z. Popović. Gameplay analysis through state projection. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 1–8. ACM, 2010.

[2] T. J. Green, M. Aref, and G. Karvounarakis. Logicblox, platform and language: a tutorial. In *Datalog in Academia and Industry*, pages 1–8. Springer, 2012.

[3] J. H. Kim, D. V. Gunn, E. Schuh, B. Phillips, R. J. Pagulayan, and D. Wixon. Tracking real-time user experience (true): a comprehensive instrumentation solution for complex systems. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 443–452. ACM, 2008.

[4] Y.-E. Liu, E. Andersen, R. Snider, S. Cooper, and Z. Popović. Feature-based projections for effective playtrace analysis. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, pages 69–76. ACM, 2011.

[5] Y.-E. Liu, T. Mandel, E. Butler, E. Andersen, E. OâĂŹRourke, E. Brunskill, and Z. Popovic. Predicting player moves in an educational game: A hybrid approach.

[6] B. Medler, M. John, and J. Lane. Data cracker: developing a visual game analytic tool for analyzing online gameplay. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, pages 2365–2374. ACM, 2011.

[7] E. O'Rourke, E. Butler, Y.-E. Liu, C. Ballweber, and Z. Popović. A deep analysis of the effects of age on in-game behavior. In *FDG '13: Proceedings of the Eighth International Conference on the Foundations of Digital Games*, 2013.

[8] A. Smith, E. Butler, and Z. Popović. Quantifying over play: Constraining undesirable solutions in puzzle design. In *FDG '13: Proceedings of the Eighth International Conference on the Foundations of Digital Games*, 2013.

[9] A. M. Smith, E. Andersen, M. Mateas, and Z. Popović. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 156–163, New York, NY, USA, 2012. ACM.

[10] G. Wallner and S. Kriglstein. A spatiotemporal visualization approach for the analysis of gameplay data. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, pages 1115–1124. ACM, 2012.

[11] W. White, B. Sowell, J. Gehrke, and A. Demers. Declarative processing for computer games. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 23–30. ACM, 2008.

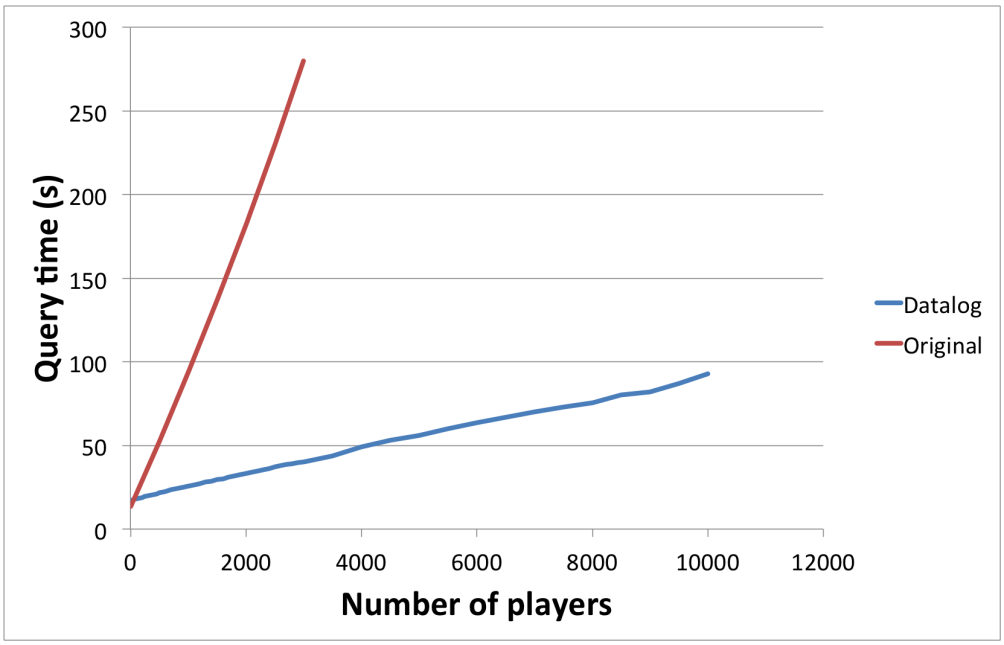

Figure 2: Computing the *wrong sinks* query for level 50.

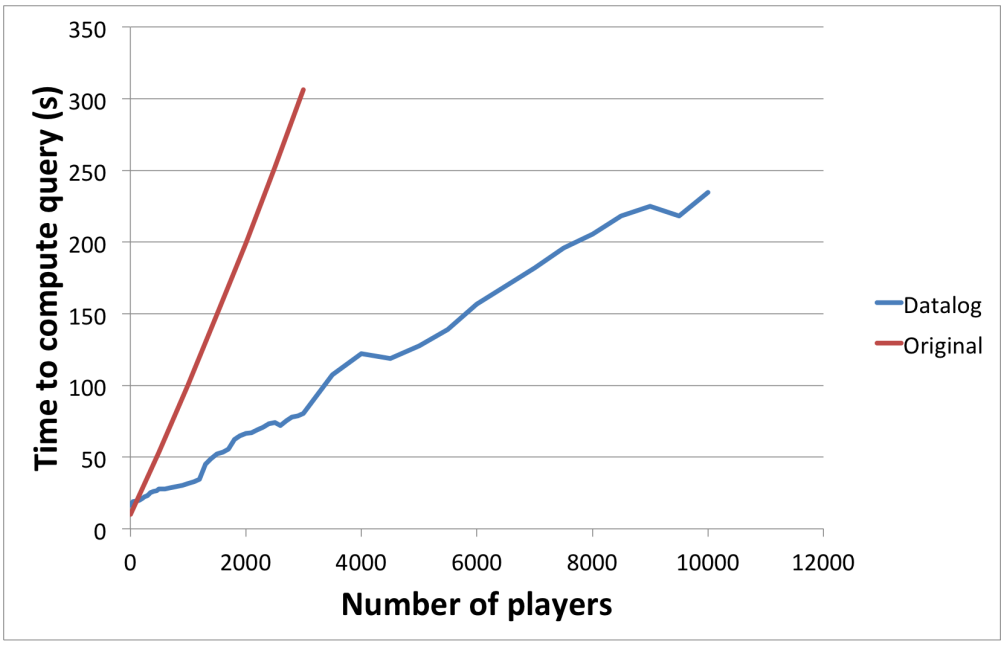

Figure 3: Computing the *wrong sinks* query for level 53.

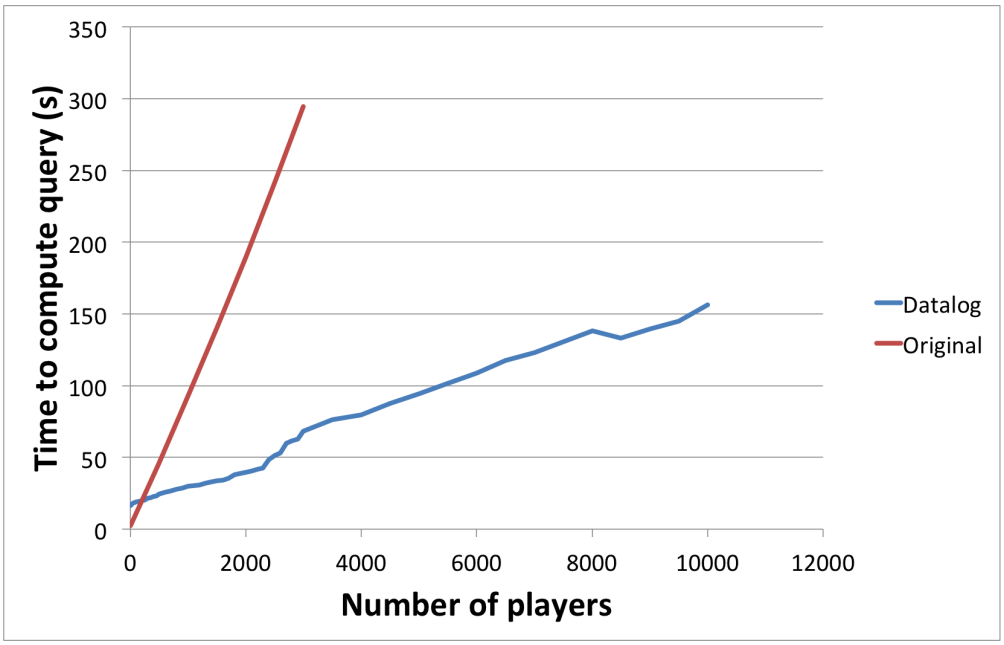

Figure 4: Computing the *wrong sinks* query for level 54.

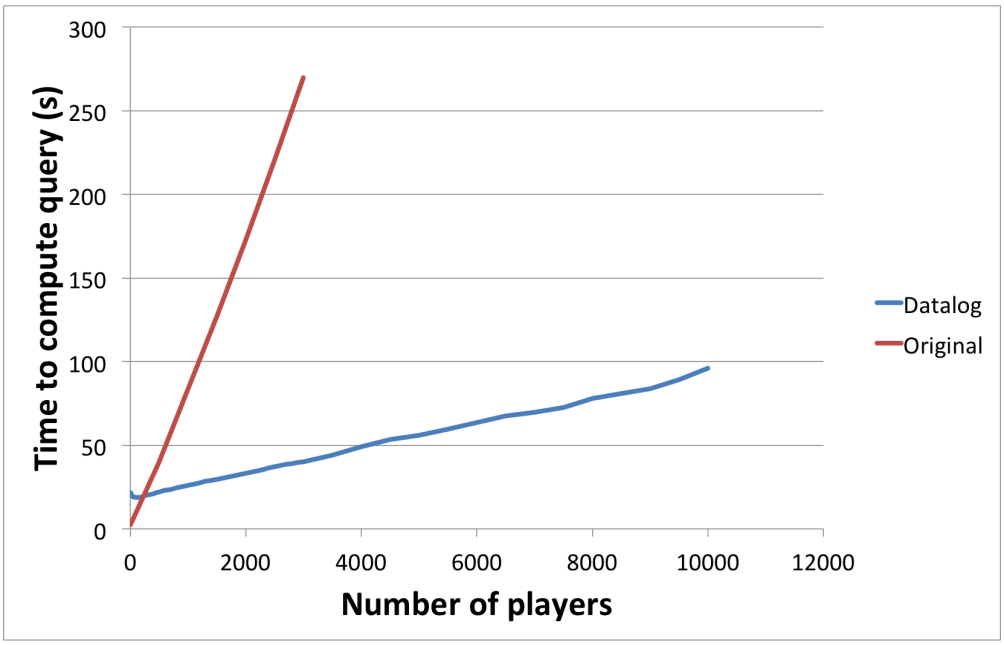

Figure 5: Computing the *bad splitters* query for level 50.

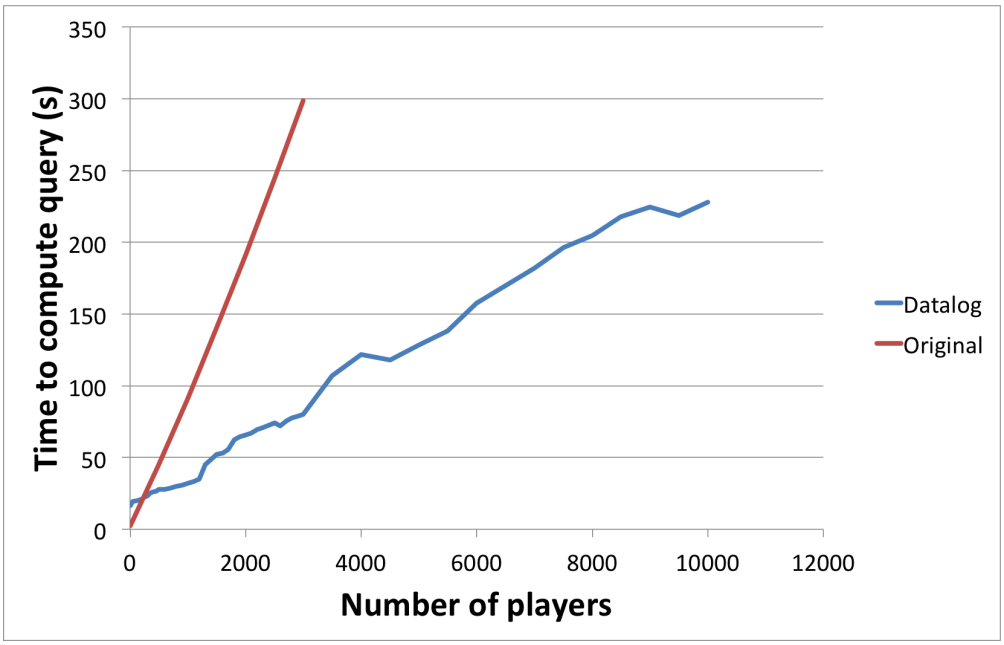

Figure 6: Computing the *bad splitters* query for level 53.

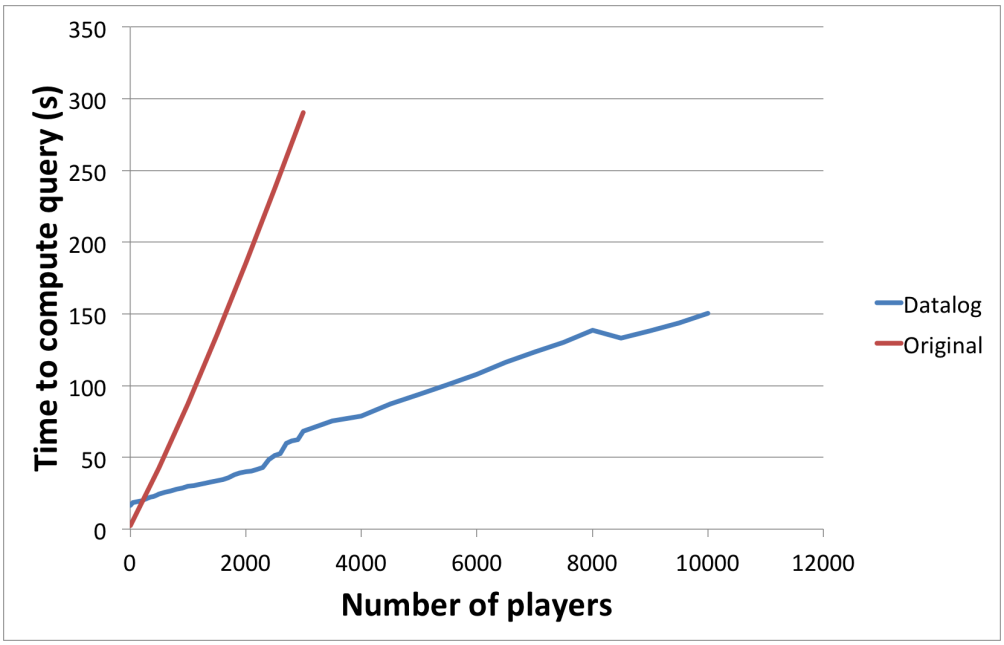

Figure 7: Computing the *bad splitters* query for level 54.

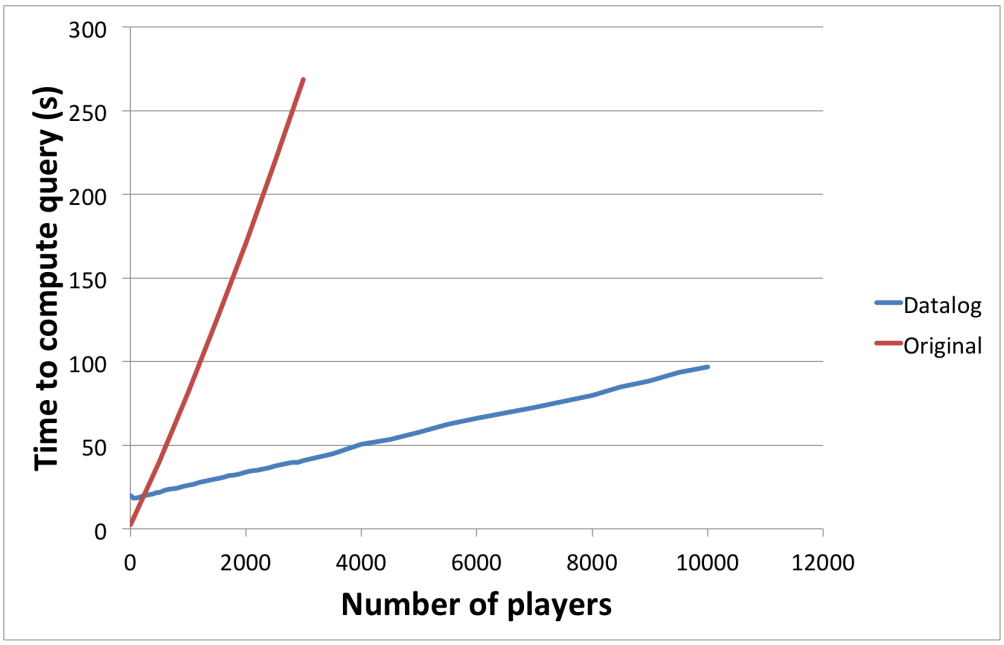

Figure 8: Computing the *proportion unique* query for level 50.

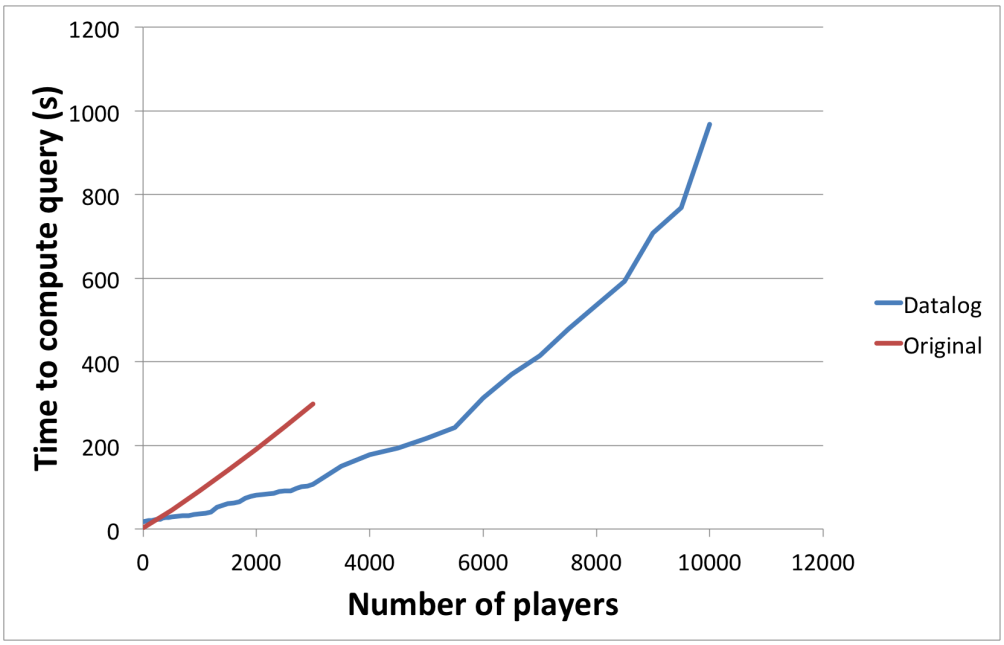

Figure 9: Computing the *proportion unique* query for level 53.

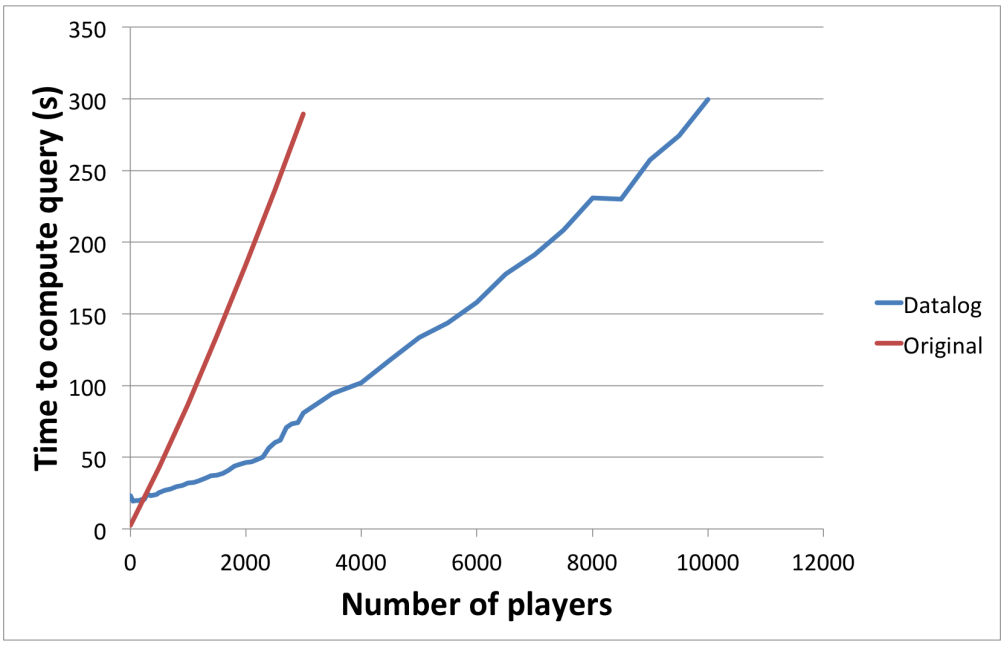

Figure 10: Computing the *proportion unique* query for level 54.