Tuning the Lookahead Distance for PostgreSQL Asynchronous IO

Wentao Wu[†], Jiasheng Hu[‡], Manoj Syamala[†], Andres Freund[†], Vivek Narasayya[†]
[†]Microsoft Corporation, Redmond, USA [‡]University of Toronto, Toronto, Canada
[†]{wentao.wu, manojsy, andres.freund, viveknar}@microsoft.com [‡]jasonhu@cs.toronto.edu

ABSTRACT

PostgreSQL (PG) recently introduced support for asynchronous IOs (PG-AIO). Under the hood, the PG-AIO subsystem adopts a classic producer/consumer model. The producer submits AIO requests to a global queue resident in shared memory, and the consumer, which by default includes a number of "IO workers," grabs the AIO requests from the queue and performs the actual IO work. To hide the implementation details of this producer/consumer model, PG offers a "read stream" abstraction with a set of API functions for table access methods, such as *sequential scan* and *bitmap heap scan*, that rely on the producer to send in AIO requests. Although this streaming API significantly simplifies the implementation of the access methods, it poses a new challenge that is performance critical: every read stream needs to decide the "lookahead distance," i.e., *the number of AIO requests to submit at the same time*. In this paper, we address this challenge by proposing an adaptive approach to automatically adjust the lookahead distance by only leveraging feedback information based on the IO completion time. Our approach is inspired by and draws a connection to the theory of *optimal flow control* in computer networking, which has a profound impact on the design of modern TCP congestion control mechanisms such as BBR. Experimental evaluation with a variety of real-world setups shows that our approach can often achieve the optimal lookahead distance that minimizes query latency.

PVLDB Reference Format:
Wentao Wu, Jiasheng Hu, Manoj Syamala, Andres Freund, Vivek Narasayya. Tuning the Lookahead Distance for PostgreSQL Asynchronous IO. PVLDB, 19(12): 3982-3995, 2026.
doi:10.14778/3827998.3828010

1 INTRODUCTION

PostgreSQL (PG) recently introduced support for asynchronous IOs (AIO) [58]. It offers a configurable system parameter `io_method` to the user with three options [59]: (1) *worker*, (2) *io_uring*, and (3) *sync*. The default option for AIO is *worker*, as *io_uring* requires special Linux kernel support [79] that is not always available, and *sync* simply reverts to synchronous IO. Therefore, we focus our discussion on the default option *worker* in this paper.

Under the hood, the PG AIO subsystem follows a producer-consumer architecture, as shown in Figure 1. At a high level, the *producer* is responsible for submitting IO requests to a *global queue* that is resident in shared memory. The *consumer* employs a number of *IO workers* (specified by the configurable system parameter `io_workers` [59]). The next available IO worker takes the IO request from the head of the queue and performs the corresponding IO operation. When it finishes, it notifies the producer that the IO request has been completed. Note that the producer is *not* blocked after submitting the IO request, unlike in the case of synchronous IO. It can continue to execute until it has to process a page contained in an IO request *that is submitted but has not yet been completed.*

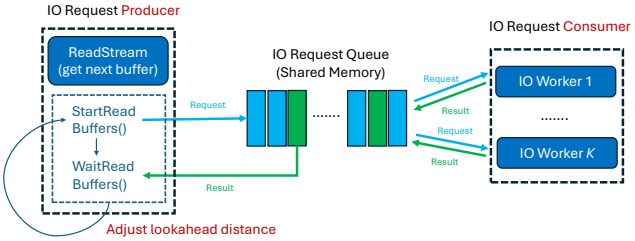


Figure 1: The architecture of the PG AIO subsystem.

To hide the details of the implementation of the PG AIO subsystem, the producer is encapsulated with an abstraction of *read stream*. It offers the following three API functions:

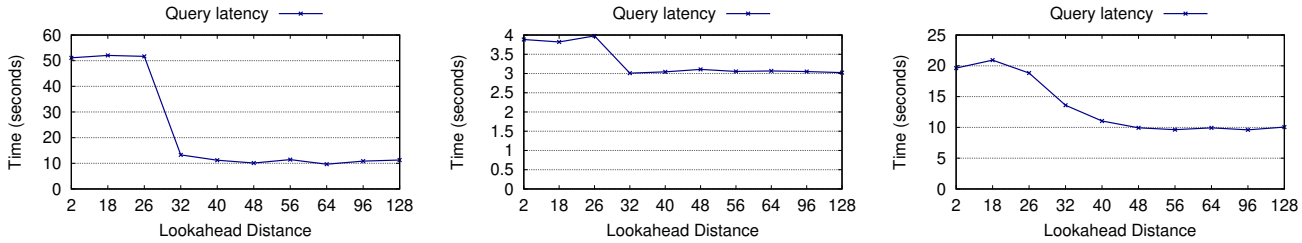
- `Begin()`, which initializes the states of the stream;
- `NextBuffer()`, which obtains the next page from the stream;
- `End()`, which cleans up the states of the stream.

Suppose that a query wants to scan a table/relation R with N disk pages. With the API, it can be simply expressed as follows:

```
ReadStream s = new ReadStream(R);
s.Begin(); // Begin the read stream.
for (int i = 0; i < N; ++i) {
   Buffer b = NextBuffer(); // Fetch the next page.
   ProcessBuffer(b);  // Perform CPU work.
}
s.End(); // End the read stream.
```

Note that the caller of the read stream (e.g., the table scan query here) needs to specify a callback that returns blocks that are needed in the future. The function `NextBuffer()` works as follows (Figure 1). It first submits one or more IO requests by calling the function `StartReadBuffers()`. It then performs some bookkeeping CPU work before hitting the function `WaitReadBuffers()`. There are two possible cases. If the (first) requested IO has been completed, then the read stream can continue to fetch the corresponding disk pages from the buffer pool, and the caller of the read stream can perform CPU work to process the pages. Otherwise, the IO is still *in progress* and the read stream will be *blocked* to *wait* for the IO to be completed. After `WaitReadBuffers()` returns, the read stream will proceed to adjust the *lookahead distance*, which controls the number of disk pages and thus IO requests to be submitted next.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828010



(a) `pg_prewarm` (Q1 with sequential IOs) (b) *Bitmap heap scan* (Q2, with fewer random IOs) (c) *Bitmap heap scan* (Q3, with more random IOs)

Figure 2: Impact of the lookahead distance on query latency (in seconds) for IO-dominant queries.

Algorithm 1: PG strategy to adjust lookahead distance

Input: d, the lookahead distance of PG AIO subsystem; $d_{\max}$, the maximum lookahead distance allowed.

```
1 Initialize d ← 1 in Begin();
2 if d < d_max then
3 |   d ← d × 2 upon any IO wait;
4 |   d ← min{d, d_max};
```

1.1 Impact of Lookahead Distance

The lookahead distance plays a critical role in the performance of the PG AIO subsystem. If the distance is too small, the available IO bandwidth can be *underutilized*. On the other hand, if the distance is too large, the IO subsystem can be *overwhelmed* with queues built up; moreover, it may pin more buffer pages than necessary in the memory, which wastes valuable memory resource that could have been used for other purposes (e.g., to serve other queries).

To demonstrate the impact of the lookahead distance on IO performance, Figure 2 presents the results of an empirical study for queries with latency dominated by IO (see Section 6.1 for the details of our evaluation setups). Specifically, we used the largest `lineitem` table from the TPC-H benchmark database with scaling factor 10. We compose the following three queries:

(Q1) `SELECT pg_prewarm('lineitem');`
(Q2) `SELECT * FROM lineitem WHERE <p1>;`
(Q3) `SELECT * FROM lineitem WHERE <p2>.`

Here, `pg_prewarm` in Q1 is a PG utility that loads relation data into the PG buffer cache [60]. It involves only sequential IOs if none of the disk pages from that relation has been brought into memory. Moreover, `pg_prewarm` requires almost no CPU work. On the other hand, `<p1>` in Q2 and `<p2>` in Q3 are two filter predicates over the column `l_shipdate`, where `<p1>` is highly selective with a small number of random IOs and `<p2>` is less selective with a large number of random IOs. We disable both *sequential scan* and *index scan* to force a query plan with *bitmap heap scan* for both Q2 and Q3. Since the *bitmap heap scan* operator requires an underlying *bitmap index scan*, we also created an index on top of column `l_shipdate`.

We have the following observations from Figure 2. First, it is clear that the query latency decreases as we increase the lookahead distance. Second, the *smallest distance* that minimizes the query latency is different for different queries. For example, the smallest distance is around 64 for Q1 as shown in Figure 2(a), 32 for Q2 as shown in Figure 2(b), and 56 for Q3 as shown in Figure 2(c). Although increasing the distance in general does not increase the query latency, it will pin more buffer pages than necessary and therefore waste memory. Therefore, our goal is to *find the smallest distance that can minimize the query latency for IO-dominant queries.*

1.2 PG Strategy for Distance Tuning

Algorithm 1 presents the strategy used by PG 18 to adjust the lookahead distance. It initializes the distance to 1 when the read stream is created and *doubles* the distance whenever the read stream needs to wait for an IO request to be completed, until the distance reaches its allowed maximum $d_{\max}$. In the current PG implementation, $d_{\max}$ is determined by the following configurable system parameters [59]:

- `shared_buffers` (SB), which specifies the amount of memory that the database server uses for shared memory buffers;
- `effective_io_concurrency` (EIC), which is the number of concurrent IOs that can be executed simultaneously;
- `io_combine_limit` (IOCL), which sets the largest number of *contiguous* disk pages that can be combined as a single IO.

More specifically, $d_{\max}$ is set as

$$d_{\max} = \min\{f(\text{SB}), \text{EIC} \times \text{IOCL}\}, \tag{1}$$

where f is some function of SB specifying the maximum number of buffer pool pages (based on the size of the SB) that can be pinned by a PG backend. The default values for EIC and IOCL are 64 and 16, respectively. By default, SB is set to 128MB but is typically configured to a much larger size—it is common practice to set SB to be at least 25% of the total database system memory for good performance. Therefore, in practice, we usually see $d_{\max} = \text{EIC} \times \text{IOCL} = 64 \times 16 = 1,024$ by Equation 1. As a result, the PG strategy in Algorithm 1 will hit $d_{\max} = 1,024$ after only 10 IO waits, which happens for all Q1, Q2, and Q3. Clearly, this distance is much larger than necessary, as Figure 2 shows. We note that PG also decreases distance by one upon any buffer pool hit, which cannot happen for the IO-dominant queries studied in this work.

1.3 Challenges and Contributions

Choosing the smallest lookahead distance without increasing query latency is therefore a nontrivial problem. The default "state agnostic" strategy used by PG (Algorithm 1) can easily overshoot the target distance. We argue that a more effective control mechanism should *monitor* the state of the PG AIO subsystem. Intuitively, if the IO subsystem is busy, we should reduce the distance to avoid further stressing it. On the other hand, if the IO subsystem is idle, we should increase the distance to better utilize its bandwidth. However, the requirement of monitoring the state of the PG AIO subsystem poses new technical challenges. First, the adjustment of the lookahead distance must be done *inside* of the read stream, while the IO operations are performed by the IO workers *outside* of the read stream. This makes it difficult to accurately and cheaply track common performance metrics, such as throughput and latency, by staying inside the read stream. Second, even if the measurements are accurate,

they typically exhibit large variation in practical storage systems commonly used for PG deployment, especially for systems that run in the cloud. As a result, even a well-designed control mechanism can be easily misled by measurements with high variance.

With these challenges in mind, in this paper we propose a novel feedback-based control mechanism for the lookahead distance, inspired by the theory of *optimal flow control* that is well established in computer networking [39–41]. In a nutshell, the *optimal operational point* of a system is equivalent to the point that maximizes the *power function* of the system, which is defined as the ratio between throughput and latency [39]. To overcome the challenge that accurate measurements of throughput and latency are difficult to obtain in the context of PG AIO, we instead design a *surrogate power function* (SPF) that is defined as the ratio between the "observed IO throughput" (which is a *lower bound* of the actual IO throughput) and the lookahead distance, and we reduce the problem of finding the smallest distance that minimizes query latency to the problem of *maximizing the SPF*. We then look for *stationary points* of the SPF that can achieve its maximum, assuming that the SPF is concave. Moreover, to overcome the challenge of large variation in the "observed IO throughput" at a fixed distance, we further propose a robust approach using *confidence intervals*. Specifically, we view the "observed IO throughput" at a given distance as a random variable rather than a constant. Instead of moving to the next larger distance immediately after an IO wait, as done by the current PG strategy (Algorithm 1), we stay at the same distance for more IO waits until the confidence interval (CI) of the random variable becomes small enough. We then return the mean as a point estimate of the random variable. Based on our experimental evaluation, it only requires a small number of sample data points to tighten the CI.

In summary, this paper makes the following contributions.

- We propose a novel approach to the problem of finding the smallest lookahead distance without increasing query latency, by following principles from the "optimal flow control" theory in computer networking with solid theoretical foundations.
- We overcome the challenge of inaccurate throughput and latency measurements with a novel surrogate power function.
- We improve the robustness of the proposed approach by using confidence intervals to overcome the challenge of large variations in the measurements.
- Our experimental evaluation results demonstrate both the efficacy and robustness of our proposed approach, which significantly outperforms the current PG strategy in Algorithm 1 as well as other baseline approaches.

There has been extensive work on a related but different problem of *prefetching* for database and storage systems [14–16, 20, 30, 45, 46, 67, 75, 83, 86]. One key challenge in the prefetching problem is to speculate or predict the pages to be fetched next based on historic access patterns. We do not face this challenge in the problem of tuning the lookahead distance, where the pages to be fetched are *known*, and we need to minimize the time to fetch them. To the best of our knowledge, this is a new problem that has not yet been studied. Although we focus on the AIO subsystem of PostgreSQL, the approaches proposed and studied in this work are potentially useful for other database systems that support asynchronous IOs.

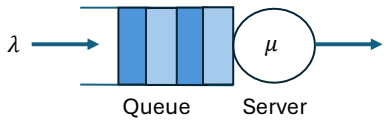


Figure 3: Example of a basic queuing system.

Paper Organization. We start with an overview of the theory of optimal flow control from computer networking (in particular, the power function) and outline its connection to the problem of tuning the lookahead distance for the PG AIO subsystem (Section 2). We then present our proposed solution based on maximizing a surrogate of the power function of the system (Section 3), as well as a set of refinement and optimization techniques (Section 4). We also study a couple of alternative solutions, which serve as baselines in our experimental evaluation (Section 5). We report the evaluation results in Section 6, discuss future directions in Section 7, summarize related work in Section 8, and conclude the paper in Section 9.

2 THEORY OF OPTIMAL FLOW CONTROL

We provide a brief overview of the "optimal flow control" theory from computer networking and discuss its connection with the problem of adjusting the lookahead distance for the PG AIO system.

2.1 Queuing Systems

It is well known that computer systems can be modeled as *queuing systems* [38, 42]. Figure 3 presents an example of a basic queuing system with the following parameters: (1) λ, the *arrival rate* that represents the number of arriving customers per unit time; (2) μ, the *service rate* that represents the number of customers that can be served per unit time by the server; (3) γ, the *throughput* of the queuing system that represents the number of completed customers per unit time; and (4) T, the average *latency* of the queuing system, which includes the service time and the waiting time (in the queue).

2.2 Power Function

The *power* P of a (queuing) system is defined as the ratio between its throughput and latency [39]. Specifically, we view the (average) latency T as a function of the throughput γ, that is, $T = h(\gamma)$. We can then also define the power as a function of γ:

$$P(\gamma) = \frac{\gamma}{T} = \frac{\gamma}{h(\gamma)}. \tag{2}$$

The optimal flow control theory points out that the *optimal operational point* of the system coincides with the optimal point (γ^*, T^*) that *maximizes the power* [39, 41]. The optimal operational point is defined as the state in which the average number of customers in the system is equal to the system capacity. For example, in the context of TCP congestion control in computer networking, this means that the network link between the sender and the receiver should be *kept just full, but not fuller* [41].

2.3 Optimal Operational Point

To find the optimal operational point, we then need to find the optimal point (γ^*, T^*) that maximizes the power function $P(\gamma)$ defined in Equation 2. Taking the derivative of $P(\gamma)$ gives

$$P'(\gamma) = \frac{T - \gamma \cdot T'}{T^2} = \frac{1}{T} - \gamma \cdot \frac{T'}{T^2}.$$

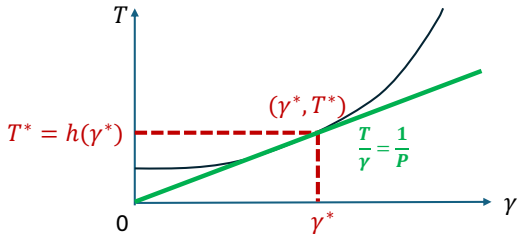


Figure 4: Illustration of the power function P and the optimal operational point (γ^*, T^*) that satisfies $\frac{T^*}{\gamma^*} = \frac{1}{P^{\max}}$.

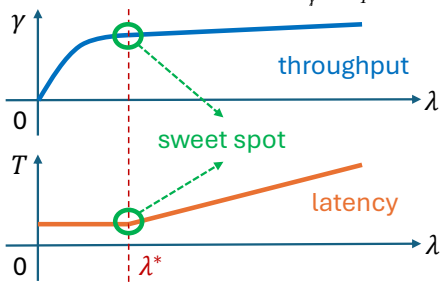


Figure 5: Illustration of the "sweet spot" in throughput and latency curves as the lookahead distance λ increases.

Setting it to zero yields $T = \gamma \cdot T'$. Using $T = h(\gamma)$ leads to $h(\gamma) = \gamma \cdot h'(\gamma)$. In other words, the *stationary point* γ^* satisfies $\gamma^* = \frac{h(\gamma^*)}{h'(\gamma^*)}$.

Theorem 1. *Assume that h is continuous, increasing, strongly convex, and twice differentiable. Then the stationary point γ^* that satisfies $\gamma^* = \frac{h(\gamma^*)}{h'(\gamma^*)}$ maximizes the power function $P(\gamma) = \frac{\gamma}{h(\gamma)}$.*

Appendix A.1 includes the proof. Figure 4 presents an example curve $T = h(\gamma)$ to illustrate the underlying intuition, where the x-axis represents throughput γ and the y-axis represents average latency T. Consider an arbitrary point (γ, T) on the curve $T = h(\gamma)$. Imagine a straight line connecting (γ, T) to the origin. The *slope* of this line is $\frac{T}{\gamma} = \frac{h(\gamma)}{\gamma} = \frac{1}{P(\gamma)}$. Therefore, the problem of maximizing power $P(\gamma)$ is equivalent to minimizing slope $\frac{1}{P(\gamma)}$. This minimum is achieved by the slope of a *tangent line* of $T = h(\gamma)$, as shown by the green line in Figure 4. As a result, the stationary point γ^* that maximizes the power function $P(\gamma^*)$ corresponds to the *tangent point* (γ^*, T^*) of the curve $T = h(\gamma)$, as shown in Figure 4. This tangent point is unique if h satisfies the assumptions stated in Theorem 1. It is also easy to verify that the *slope* of the tangent line connecting the tangent point to the origin is $\frac{T^*}{\gamma^*} = \frac{h(\gamma^*)}{\gamma^*} = \frac{1}{P(\gamma^*)}$.

2.4 Connection with PG AIO Distance Control

In PG AIO, the role of the lookahead distance is similar to the arrival rate λ in a queuing system. Therefore, we abuse the notation by using λ to also represent the lookahead distance. When increasing λ, at the beginning throughput will increase while latency will remain stable; after certain "sweet spot," throughput will not increase much while latency will start to increase. The intuition behind the power function is that maximizing power translates to maintaining near-maximum throughput without increasing latency, which is equivalent to finding the above "sweet spot" of λ. Figure 5 illustrates this relationship between the lookahead distance and the "sweet spot" in the throughput and latency curves.

3 TUNING PG AIO LOOKAHEAD DISTANCE

We now apply the idea of maximizing the power function from the theory of optimal flow control in computer networking to the problem of tuning the lookahead distance for PG AIO.

A straightforward approach would be to keep track of throughput and latency and then find the optimal lookahead distance λ^* that corresponds to the "sweet spot" in the throughput and latency curves, as shown in Figure 5. However, in the context of the PG AIO subsystem, it is difficult to observe the IO latency *inside* of a read stream, given that IOs are performed by IO workers *outside* of the read stream. That is, when the read stream comes back to check whether it needs to wait on an IO submitted before, the IO may have been finished a while ago. Therefore, the read stream does not know when the IO was actually started and finished. Given the difficulty of accurately measuring IO latency, the measurement of throughput will also be inaccurate. As a result, the simple idea of relying on keeping track of throughput and latency cannot work. In the following, we propose a new approach that does not rely on accurate measurements of throughput and latency.

Our key observation is that finding the optimal distance λ^* that results in the stationary point γ^* of the power function $P(\gamma)$ does *not* require the exact representation of $P(\gamma)$. That is, some *surrogate* of the power function would be sufficient if our goal was to find a point to maximize power. In fact, using a surrogate is a well-known tactic in Bayesian optimization where the objective function to optimize is an unknown black box [50, 51, 68].

3.1 Surrogate of Power Function

We define *surrogate* of the power function $P(\gamma)$ as

$$P_s(\lambda) = \frac{\gamma}{\lambda} = \frac{g(\lambda)}{\lambda}. \tag{3}$$

Compared to the standard power function defined in Equation 2, P_s replaces the latency T in the denominator by the lookahead distance λ. Moreover, instead of being parametrized by throughput γ as in Equation 2, P_s is parametrized by λ where we view γ as a function $g(\lambda)$ of λ. We then want to maximize the surrogate power function $P_s(\lambda)$. The intuition of maximizing $P_s(\lambda)$ is clear: we want to *maximize throughput with the minimum lookahead distance*, which, as a coincidence, reiterates our motivation in the introduction of this paper on tuning the lookahead distance.

Now consider again the stationary point of $P_s(\lambda)$. Taking the derivative of $P_s(\lambda)$, we have

$$P_s'(\lambda) = \frac{\gamma' \cdot \lambda - \gamma}{\lambda^2} = \frac{\gamma'}{\lambda} - \frac{\gamma}{\lambda^2}.$$

Setting $P_s'(\lambda) = 0$ gives $\gamma' \cdot \lambda = \gamma$. Therefore, the stationary point λ^* of $P_s(\lambda)$ satisfies the following condition:

$$\lambda^* = \frac{\gamma}{\gamma'} = \frac{g(\lambda^*)}{g'(\lambda^*)}. \tag{4}$$

Theorem 2. *Assume that g is continuous, increasing, strongly concave, and twice differentiable. Then the stationary point λ^* that satisfies $\lambda^* = \frac{g(\lambda^*)}{g'(\lambda^*)}$ maximizes the surrogate $P_s(\lambda) = \frac{g(\lambda)}{\lambda}$.*

The proof of Theorem 2 can be found in Appendix A.2. Compared to Theorem 1, Theorem 2 requires that g be concave rather than convex. This is because now we view throughput as a function of the lookahead distance λ. By the example shown in Figure 5,

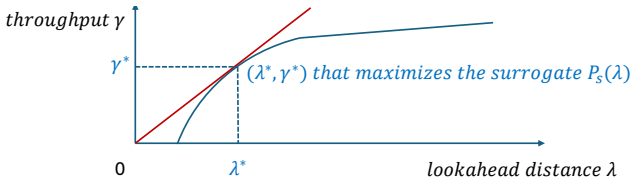


Figure 6: Illustration of the surrogate power function $P_s(\lambda)$ and its stationary point (λ^*, γ^*) that satisfies $\frac{\gamma^*}{\lambda^*} = P_s^{\max}$.

Algorithm 2: Estimation of the throughput $\gamma = g(\lambda)$

Input: $\{\lambda_j\}_{j=1}^n$, the set of increasing distances to probe; $\{m_j\}_{j=1}^n$, the number of IO waits for each distance λ_j.
Output: $\{\gamma_j\}_{j=1}^n$, the observed throughput based on Equation 5.

```
1 for 1 ≤ j ≤ n do
2     Observe m_j IO waits;
3     for 1 ≤ k ≤ m_j do
4         Compute γ_{j,k} using Equation 5;
5     γ_j ← (1/m_j) Σ_{k=1}^{m_j} γ_{j,k};
6 return {γ_j}_{j=1}^n;
```

this assumption on the concavity of $\gamma = g(\lambda)$ is reasonable. In fact, we can reparametrize P_s with γ as $P_s(\gamma) = \frac{\gamma}{g^{-1}(\gamma)}$. Since g is an increasing function, if g is concave, then g^{-1} is convex. Then we can apply Theorem 1 to conclude that $\gamma^* = g(\lambda^*)$ maximizes $P_s(\gamma)$ without the need of Theorem 2.

Figure 6 illustrates the stationary point (λ^*, γ^*) that maximizes the surrogate power function $P_s(\lambda)$. We can see a *symmetric* geometric interpretation by comparing it with Figure 4.

Remark. As a simple and intuitive example, if the latency T is a linear and increasing function of the lookahead distance λ, that is, $T = a \cdot \lambda$ with $a > 0$, then the stationary point λ^* that maximizes the surrogate $P_s(\lambda)$ also maximizes the power function $P(\gamma)$, as we have $P(\gamma) = \frac{\gamma}{T} = \frac{\gamma}{a \cdot \lambda} = \frac{1}{a} \cdot P_s(\lambda)$. That is, $P(\gamma)$ is also a linear and increasing function of $P_s(\lambda)$.

3.2 Estimation of Throughput

The throughput function $\gamma = g(\lambda)$ defined in the surrogate $P_s(\lambda)$ is *unknown*. We now propose techniques to estimate $g(\lambda)$.

Suppose that there are K observed IO waits in the read stream and that the block size is B bytes. After the k-th *IO wait* ($1 \le k \le K$) of the read stream, let N_k be *the total number of IO blocks that have been completed so far* and t_k be *the timestamp*. We set $N_0 = 0$ and $t_0 = 0$. We estimate the throughput γ_k at the k-th IO wait as

$$\gamma_k = B \cdot \frac{N_k - N_{k-1}}{t_k - t_{k-1}}. \tag{5}$$

Note that this estimated throughput is different from the actual throughput of the PG AIO system. In fact, it is a *lower bound* of the actual throughput, because there can be *ongoing* IOs.

Algorithm 2 presents a simple approach to obtain the throughput function $\gamma = g(\lambda)$. We define a set of increasing lookahead distances $\{\lambda_j\}_{j=1}^n$. For each distance λ_j, we observe m_j IO waits before moving to the next distance. For each IO wait, we use Equation 5 to compute the observed throughput. We then take the average of these m_j IO waits as the observed throughput γ_j that corresponds to the distance λ_j. In the end, we obtain n observed data points $\{(\lambda_1, \gamma_1), ..., (\lambda_n, \gamma_n)\}$ for the throughput function $\gamma = g(\lambda)$.

Algorithm 3: Computation of the stationary point λ^*

Input: $\{(\lambda_j, \gamma_j)\}_{j=1}^n$, the pairs of distance and throughput.
Output: λ^*, the stationary point that maximizes $P_s(\lambda)$.

```
1  for 1 ≤ j ≤ n do
2      f(λ_j) ← λ_j · g'_j − γ_j; // Compute g'_j using Equation 7.
3      if f(λ_j) is zero then
4          # Case 1: there is a root.
5          return λ_j;
6  for 1 ≤ j ≤ n − 1 do
7      if f(λ_j) · f(λ_{j+1}) < 0 then
8          # Case 2: there is a "change of signs."
9          return λ ← argmin_{λ∈(λ_j, λ_{j+1})} |f(λ)|;
10 # Case 3: there is no root or "change of signs."
11 J ← argmin_{1≤j≤n} |f(λ_j)|;
12 return λ ← argmin_{λ∈(λ_{J−1}, λ_{J+1})} |f(λ)|;
```

Note that we need to specify the two input parameters for Algorithm 2: (1) $\{\lambda_j\}_{j=1}^n$, the set of increasing distances to probe; and (2) $\{m_j\}_{j=1}^n$, the number of IO waits for each distance λ_j. We use a *geometric* distance probing scheme for (1) with $\lambda_{j+1} = \theta \cdot \lambda_j$, where $\theta > 1$ is some *pacing factor*. We set $\theta = 1.5$ in our experimental evaluation (Section 6). The problem of specifying (2) is more complicated due to the variation that we observed in the throughput measurements, and we discuss this problem in more detail with our solution in Section 4.1.

3.3 Computation of the Stationary Point

To compute the stationary point λ^*, we need to solve the equation

$$\lambda \cdot g'(\lambda) = g(\lambda). \tag{6}$$

Since we do not have the analytic form of $g(\lambda)$, we need to solve this equation with a numerical procedure. Since g is the throughput function that fits the collected data points $\{(\lambda_1, \gamma_1), ..., (\lambda_n, \gamma_n)\}$, we can approximate its derivative g' as

$$g'_j = \frac{\gamma_j - \gamma_{j-1}}{\lambda_j - \lambda_{j-1}}, \text{ for } 1 \le j \le n. \tag{7}$$

We define $\lambda_0 = 0$ and $\gamma_0 = 0$ so that all derivatives are well defined.

Let $f(\lambda)$ be the following function

$$f(\lambda) = \lambda \cdot g'(\lambda) - g(\lambda).$$

The problem of solving Equation 6 is equivalent to finding the *root* of $f(\lambda)$. Again, we need to compute the root numerically, since we do not have an analytical form of g and thus f.

Note that we have $f'(\lambda) = (\lambda \cdot g'(\lambda))' - g'(\lambda) = g'(\lambda) + \lambda \cdot g''(\lambda) - g'(\lambda) = \lambda \cdot g''(\lambda)$. If g is strongly concave, $g''(\lambda) < 0$. As a result, $f'(\lambda) < 0$ and thus f is strictly decreasing. Therefore, if $f(\lambda)$ has a root, then the root will be *unique* in this case. Of course, the real g based on the observed data points may not be strongly concave, so it is possible that we cannot find a root for f or there are multiple roots for f. In the former case, we instead return the minimum distance λ that minimizes $|f(\lambda)|$. In the latter case, we return the minimum root that corresponds to the minimum distance. Algorithm 3 presents the details of this procedure.

Specifically, we first compute $f(\lambda_j)$ for each λ_j with $1 \le j \le n$ (line 2). If we find any λ_j with $f(\lambda_j) = 0$, then we return λ_j as the stationary point (line 5). On the other hand, if there is no λ_j satisfying $f(\lambda_j) = 0$, we then check if there is any "change of

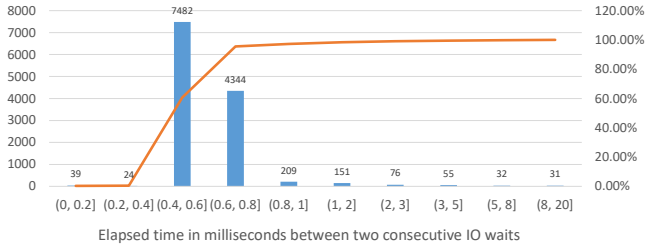


Figure 7: Variation of observed Δt_k on cloud storage system. The coefficient of variation (CV) is defined as the ratio between the standard deviation and the mean. We observed a CV of 1.22 (with mean 0.67 ms and standard deviation 0.82 ms), which implies a huge variation of the measurements.

signs," that is, if there exist λ_j and λ_{j+1} such that $f(\lambda_j) \cdot f(\lambda_{j+1}) < 0$. If so, we search within the range $(\lambda_j, \lambda_{j+1})$ and return the λ that minimizes $|f(\lambda)|$ (line 9). Finally, it is possible that we fail to find any root and in that case we look for λ_J that minimizes $\{|f(\lambda_j)|\}_{j=1}^{n}$ (line 11). We refine the result by searching within the range $(\lambda_{J-1}, \lambda_{J+1})$ to find a distance λ that further reduces $|f(\lambda)|$.

4 REFINEMENTS AND OPTIMIZATIONS

In this section, we present a set of refinement and optimization techniques to improve the framework detailed in Section 3 for tuning the lookahead distance. There are several practical issues that hinder the direct applicability of the framework.

First, Algorithm 2 requires specifying the number of IO waits at each distance, which is a nontrivial problem given the variation or uncertainty observed in the throughput measurements. We propose to view the IO waits as "online samples" and use confidence intervals (CIs) based on the central limit theorem (CLT) to decide the stopping criteria of this sampling procedure (Section 4.1).

Second, Algorithm 3 looks for "change of signs" to check the existence of stationary points. However, not all changes of signs are worth considering. We propose to exploit the convexity and concavity properties of the throughput-distance curve $\gamma = g(\lambda)$ to remove stationary points that are valid but misleading (Section 4.2).

In addition to these two major optimizations, we also include other minor refinements in Section 4.3.

4.1 Reduction of Uncertainty

The estimated throughput based on Equation 5 requires two quantities: (1) $\Delta N_k = N_k - N_{k-1}$ and (2) $\Delta t_k = t_k - t_{k-1}$. In practice, we found that Δt_k often exhibits a high variation. That is, for a fixed lookahead distance and a stable number of IO blocks read in (i.e., fixed ΔN_k), Δt_k can be very different for different k. This is true regardless of whether we measure Δt_k within the read stream (i.e., the producer in Figure 1) or within one of the IO workers (i.e., the consumer in Figure 1). As a result, the estimated throughput also exhibits a high variation. This is especially the case when the PostgreSQL database is deployed on top of cloud storage systems, e.g., Azure managed disks [53].

Figure 7 presents a concrete example to show the observed variation of Δt_k. It is a histogram that represents the distribution of the observed Δt_k. We observe that the distribution is quite long-tail, covering a wide range between 0 and 20 milliseconds. Specifically, let $\mu_j = \frac{1}{m_j}\sum_{k=1}^{m_j} \Delta t_{j,k}$ and $\sigma_j^2 = \frac{1}{m_j}\sum_{k=1}^{m_j}(\Delta t_{j,k} - \mu_j)^2$ be the mean

Algorithm 4: Estimation of μ_j based on confidence interval

Input: λ, the lookahead distance to probe; m_0, the number of samples for a "pilot run"; $0 < r < 1$, the target relative error; $0 < \alpha < 1$, the significance level.
Output: $\bar{\gamma}$, the mean of the estimated throughput.

1 $(\bar{\gamma}_0, s_0) \leftarrow$ `PilotRun`(m_0);
2 $z \leftarrow z_{1-\alpha/2}$, $M \leftarrow \left(\frac{z \cdot s_0}{r \cdot \bar{\gamma}_0}\right)^2$; // Equation 9
3 **for** $1 \le m \le M$ **do**
4 Observe the next IO wait with distance λ;
5 Compute the estimated throughput $\hat{\gamma}$ using Equation 5;
6 Update the mean $\bar{\gamma}$ and the standard deviation s with $\hat{\gamma}$;
7 **if** $z_{1-\alpha/2} \cdot \frac{s}{\sqrt{m}} \le r \cdot \bar{\gamma}$ **then**
8 **Break**;
9 **return** $\bar{\gamma}$;

and variance (where σ_j stands for the standard deviation) of the distribution of $\Delta t_{j,k}$. The "coefficient of variation (CV)" is defined as $\mathrm{CV}_j = \frac{\sigma_j}{\mu_j}$. A high CV then implies a large variation. For the example in Figure 7, we observe a CV of 1.22, namely, the standard deviation is 1.22× of the mean. This high variation indicates a high uncertainty of μ_j and therefore the observed throughput γ_j.

We reduce the uncertainty with an approach based on "confidence interval (CI)" of the observed Δt_k and thus γ_k. This approach does not require any explicit assumption of the underlying distribution (e.g., normal or exponential distribution) and only relies on the *central limit theorem* (CLT). The downside of this approach is that it may require more observations to converge compared to assuming explicit distributions.

Suppose that the observed throughput γ follows a distribution with mean μ and standard deviation σ that are both unknown. Our goal is to estimate μ with m i.i.d. samples γ_1, ..., γ_m from the distribution. Applying the CLT, we have $z = \frac{\bar{\gamma}-\mu}{\sigma/\sqrt{m}} \sim \mathcal{N}(0,1)$, where $\bar{\gamma}$ is the mean of m samples and $\mathcal{N}(0,1)$ is the *standard normal distribution*. When m is relatively large (e.g., $m \ge 30$), we can use the standard deviation of the samples s to replace the unknown σ. As a result, we have $z = \frac{\bar{\gamma}-\mu}{s/\sqrt{m}} \sim \mathcal{N}(0,1)$. The confidence interval (CI) is then defined as $[\bar{\gamma} - z \cdot \frac{s}{\sqrt{m}}, \bar{\gamma} + z \cdot \frac{s}{\sqrt{m}}]$, where z is the z-score that is determined by the *confidence level*, i.e. the length of the CI, $1-\alpha$. Here, α is the *significance level* in hypothesis testing. For example, $\alpha = 0.05$ means a confidence level of 0.95, which results in a z-score $z = z_{1-\alpha/2} = 1.96$. That is, the probability that the mean μ is covered by the confidence interval $[\bar{\gamma} - 1.96 \cdot \frac{s}{\sqrt{m}}, \bar{\gamma} + 1.96 \cdot \frac{s}{\sqrt{m}}]$ is 0.95. $\hat{\mu} = \bar{\gamma}$ is then a reliable estimator of μ when the CI is tight.

CI-based Online Sampling. Let r be some tolerable *relative error* of $\hat{\mu} = \bar{\gamma}$ with $0 < r < 1$, that is, $(1-r) \cdot \hat{\mu} \le \mu \le (1+r) \cdot \hat{\mu}$. Equivalently, we have $-r \cdot \hat{\mu} \le \hat{\mu} - \mu \le r \cdot \hat{\mu}$, that is, $-r \cdot \bar{\gamma} \le \bar{\gamma} - \mu \le r \cdot \bar{\gamma}$. It follows that $|\bar{\gamma} - \mu| \le r \cdot \bar{\gamma}$. Given some CI with length $1-\alpha$, we have $|\bar{\gamma} - \mu| \le z_{1-\alpha/2} \cdot \frac{s}{\sqrt{m}}$. As a result, it requires

$$z_{1-\alpha/2} \cdot \frac{s}{\sqrt{m}} \le r \cdot \bar{\gamma}. \tag{8}$$

This suggests an online sampling procedure that monitors the stopping condition stated by Equation 8. One problem is that $\bar{\gamma}$ and s in Equation 8 depend on the sample size m. To have some control over the sample size to make the online sampling procedure more stable, we estimate m using a *pilot run*.

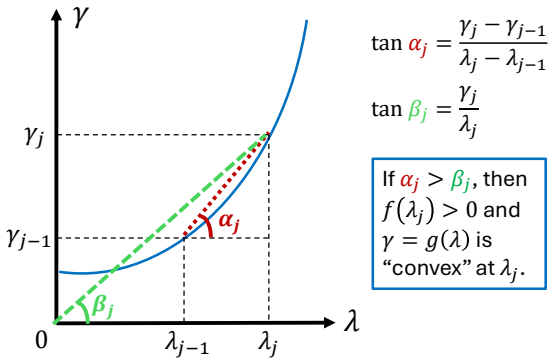


Figure 8: Illustration of the connection between the convexity of $\gamma = g(\lambda)$ and $f(\lambda) > 0$ at the distance λ_j.

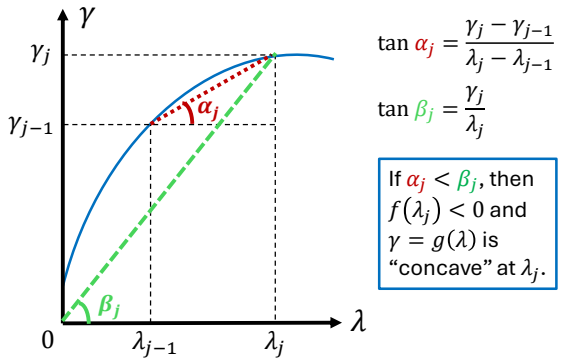


Figure 9: Illustration of the connection between the concavity of $\gamma = g(\lambda)$ and $f(\lambda) < 0$ at the distance λ_j.

Pilot Run. Equation 8 suggests that we need a sample size m with $m \geq \left(\frac{z_{1-\alpha/2} \cdot s}{r \cdot \bar{\gamma}}\right)^2$. However, we do not know $\bar{\gamma}$ and s before online sampling begins. Our solution is to use a small number (e.g. $m_0 = 30$) of samples to obtain their mean $\bar{\gamma}_0$ and standard deviation s_0 as estimates for $\bar{\gamma}$ and s. This gives us an estimate M for m as

$$M = \left(\frac{z_{1-\alpha/2} \cdot s_0}{r \cdot \bar{\gamma}_0}\right)^2, \tag{9}$$

where $0 < r < 1$ is the *relative error* allowed for the estimation of μ.

Putting It Together. Algorithm 4 presents the details of this CI-based "online sampling" approach. We start with a "pilot run" described above to determine an upper bound M on the number of samples, i.e., IO waits, to take (line 1). For each incoming IO wait, we compute the estimated throughput $\hat{\gamma}$ using Equation 5, and then update $\bar{\gamma}$ and s with $\hat{\gamma}$. We check whether the stopping condition in Equation 8 is met (line 7) and exit online sampling if so.

Discussion. The "samples" are not necessarily i.i.d., as they depend on the characteristics of the IOs from the read stream. For example, a read stream with sequential reads can be quite different from a read stream with random reads. Sometimes, a read stream can contain IO *bursts* that are difficult to predict or capture by Algorithm 4 if the bursts come after the first M IO waits. This can reduce the efficacy of applying the CLT, which assumes i.i.d. samples (there are variants of the standard CLT that do not require i.i.d. samples, but they require other assumptions). We leave the investigation of the impact of these subtleties for future work.

4.2 Exploitation of Convexity

Algorithm 3 looks for "change of signs" to locate potential stationary points. Specifically, it checks the condition $f(\lambda_j) \cdot f(\lambda_{j+1}) < 0$. However, not all such distances are worth considering, depending on the convexity and concavity properties of the curve $\gamma = g(\lambda)$.

Figure 8 illustrates the case when $\gamma = g(\lambda)$ is *convex* at λ_j. We define the angle α_j that satisfies $\tan \alpha_j = \frac{\gamma_j - \gamma_{j-1}}{\lambda_j - \lambda_{j-1}}$ and the angle β_j that satisfies $\tan \beta_j = \frac{\gamma_j}{\lambda_j}$. If $\gamma = g(\lambda)$ is convex at λ_j, then $\alpha_j > \beta_j$ and thus $\tan \alpha_j > \tan \beta_j$. As a result, $\frac{\gamma_j - \gamma_{j-1}}{\lambda_j - \lambda_{j-1}} > \frac{\gamma_j}{\lambda_j}$. That is, $g'(\lambda_j) > \frac{g(\lambda_j)}{\lambda_j}$. Therefore, $f(\lambda_j) = \lambda_j \cdot g'(\lambda_j) - g(\lambda_j) > 0$.

Similarly, Figure 9 illustrates the case when $\gamma = g(\lambda)$ is *concave* at λ_j. We have $\alpha_j < \beta_j$ instead and thus $\tan \alpha_j < \tan \beta_j$. Therefore, we can conclude $f(\lambda_j) < 0$.

Now consider the "change of signs" condition $f(\lambda_j) \cdot f(\lambda_{j+1}) < 0$ again. There are two possible cases as follows.

- $f(\lambda_j) < 0$ and $f(\lambda_{j+1}) > 0$: It implies that $g(\lambda)$ is concave at λ_j but is convex at λ_{j+1}. This case is not interesting, as it means that the growth of the throughput $\gamma_{j+1} = g(\lambda_{j+1})$ at the distance λ_{j+1} is *accelerating* (see Figure 8). That is, the growth of the throughput is faster than a projected linear function (with slope $\tan \beta_j$) of the distance. This is an indicator that the "pipe" is not full yet [41], and we can push more by increasing the distance.
- $f(\lambda_j) > 0$ and $f(\lambda_{j+1}) < 0$: It implies that $g(\lambda)$ is convex at λ_j but is concave at λ_{j+1}. This case is interesting, as it suggests that the growth of the throughput $\gamma_{j+1} = g(\lambda_{j+1})$ at the distance λ_{j+1} is *decelerating* (see Figure 9).

Therefore, we can replace the "change of signs" condition in Algorithm 3 by the second case above. That is, $f(\lambda_j) > 0$ and $f(\lambda_{j+1}) < 0$. This helps skip stationary points that belong to the first case above, which can mislead the search to stop too early with small distances that increase the IO latency.

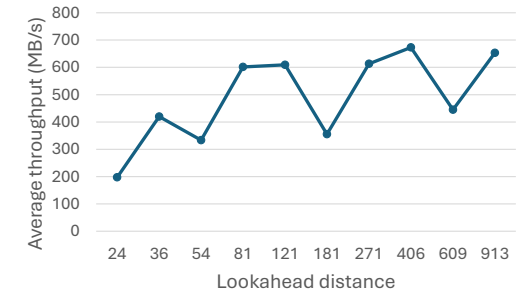
4.3 Other Refinements

4.3.1 Smoothing. We have implicitly assumed that increasing distance will not *decrease* the observed throughput. Figure 10 presents a concrete example of a throughput-distance curve $\gamma = g(\lambda)$ observed in our experimental evaluation. The original curve in Figure 10(a) contains some zig-zag patterns, indicating that the observed throughput does not always increase due to various reasons, such as instability and variation in the performance of the IO system. After smoothing, the curve becomes non-decreasing and makes it easier to look for the stationary points, as Figure 10(b) shows.

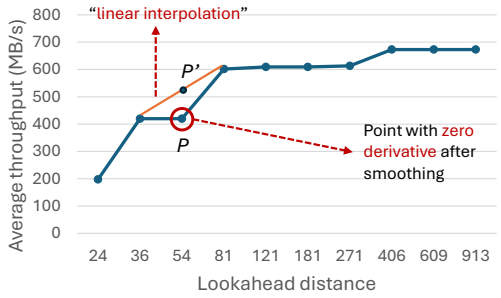
4.3.2 Linear Interpolation. One problem of this simple smoothing technique is that the derivatives at the smoothed points become zero, which is less indicative. The point P in Figure 10(b) presents an example of this problem. To address this issue, we further use a *linear interpolation* mechanism to avoid zero derivatives. The point P' in Figure 10(b) presents the "calibrated version" of P after applying the linear interpolation.

5 BASELINE SOLUTIONS

We propose a few alternative solutions based on the observed throughput function $\gamma = g(\lambda)$, which serve as baselines for a comparative study in our experimental evaluation (see Section 6).



(a) Original curve before smoothing and linear interpolation



(b) After smoothing and linear interpolation

Figure 10: Smoothing to make throughput non-decreasing.

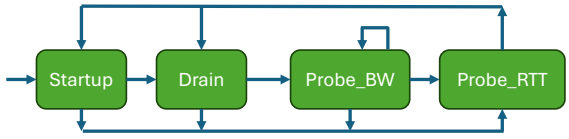


Figure 11: The state machine of BBR.

5.1 Customized BBR

BBR is a modern TCP congestion control mechanism proposed by Google [10]. It is based on the idea of maintaining the *bandwidth-delay product* (BDP) of the network link between the TCP sender and the receiver. As was pointed out in [41], in many situations, this coincides with maintaining the optimal operational point of the underlying queuing system, which is equivalent to maximizing the power function. Figure 11 presents the state machine employed by BBR. It contains the following four states [10].

- *Startup*: It ramps up by increasing the *pacing rate* of sending packets until it saturates the network bandwidth. The maximum bandwidth $B^{\max}$ observed in this process is recorded.
- *Drain*: It keeps decreasing the pacing rate to *drain* the potential queue built up during *Startup*. The minimum delay (between sending a packet and receiving the ACK) $D^{\min}$ is recorded. The BDP is computed as BDP $= B^{\max} \times D^{\min}$. The pacing rate is set to match the BDP.
- *Probe_BW*: It loops over a predefined set of multipliers, e.g. {1.25, 0.75, 1, 1, 1, 1, 1, 1}, to increase/decrease/maintain the pacing rate and monitor the corresponding maximum bandwidth observed, with the purpose of detecting a potential change in network conditions, e.g., a new link/path may have been established between the sender and receiver.
- *Probe_RTT*: It is constantly invoked, e.g. every 10 seconds, to measure the latest minimum delay $D^{\min}$. The BDP and the pacing rate are then updated accordingly with the latest measurements of $B^{\max}$ and $D^{\min}$.

Algorithm 5: Customized BBR for tuning PG AIO distance.

Input: $d_{\min}$, the minimum distance allowed; $d_{\max}$, the maximum distance allowed; $\rho > 1$, acceleration factor in the *Startup* phase; $0 < \beta < 1$, decay factor in the *Drain* phase; N, number of IO waits in the *Startup* phase without observing an increase of throughput; $0 < p < 1$, percentage of drop for observed throughput in the *Drain* phase; K, the maximum number of IO waits allowed in the *Drain* phase.

Output: λ^*, the best distance found.

Startup:
$\lambda^* \leftarrow d_{\min}$, $\gamma_{\max} \leftarrow 0$;
while $\lambda^* < d_{\max}$ **do**
  Observe the next IO wait and estimate throughput $\hat{\gamma}$ (Eq.5);
  $\gamma_{\max} \leftarrow \max\{\hat{\gamma}, \gamma_{\max}\}$;
  if $\gamma_{\max}$ *has not changed for N IO waits* **then**
    Break;
  $\lambda^* \leftarrow \rho \cdot \lambda^*$;
Drain:
for $1 \le i \le K$ **do**
  Observe the next IO wait and estimate throughput $\hat{\gamma}$ (Eq.5);
  if $\hat{\gamma} < (1-p) \cdot \gamma_{\max}$ **then**
    Break;
  $\lambda^* \leftarrow \beta \cdot \lambda^*$;
return λ^*;

As illustrated in Algorithm 5, we customize BBR in the context of tuning PG AIO distance as follows.

- *Startup*: It ramps up by increasing the *lookahead distance* until the maximum distance. Specifically, we multiply the distance by some *acceleration factor* $\rho > 1$ after an IO wait. We exit *Startup* if the observed throughput does not increase after N IO waits.
- *Drain*: To drain the potential queue of IOs built up in the *Startup* phase, it reduces the distance by multiplying it by some *decay factor* $0 < \beta < 1$ after an IO wait. We exit *Drain* if the observed throughput drops by p or after K IO waits.

We omit the two states *Probe_BW* and *Probe_RTT* for two reasons. First, storage systems are typically much more stable than network systems (e.g., there is no analogy of link/path in a storage system). Second, unlike *Startup* and *Drain*, which require only a one-time computation, both *Probe_BW* and *Probe_RTT* are invoked frequently (e.g., on every IO wait), which can significantly increase the CPU computation overhead. In our experimental evaluation (Section 6), we set $\rho = 1.5$, $N = 2$, $\beta = 0.9$, $p = 0.05$ (i.e., 5%), and $K = 5$. These settings are common in existing BBR implementations for TCP congestion control [19, 27].

5.2 Multi-armed Bandits

We can model the problem of determining the optimal lookahead distance using *multi-armed bandits*. Specifically, we view each candidate distance as an *arm*. Selecting an arm then means choosing the corresponding lookahead distance, and the observed throughput at the distance is the *reward* of the arm. We use the classic "upper confidence bound (UCB)" strategy to play the arms and return the arm (i.e. distance) with the maximum average throughput.

5.3 Bayesian Optimization

We can also address the problem of finding the optimal lookahead distance using Bayesian optimization [50, 51, 68]. The idea is to

Table 1: IO profiles of the queries used in the evaluation.

Query	Type	# Blocks	# IOs	Avg. IO Size
Q1	Sequential reads	1,125,159	70,322	16
Q2	Random reads	24,102	23,583	1.022
Q3	Random reads	138,551	131,327	1.055

view throughput as an unknown function of the distance and then apply Bayesian optimization. In particular, we use GP-EI [68] in this work for its known efficacy, which uses Gaussian process (GP) with expected improvement (EI) as the acquisition function.

6 EXPERIMENTAL EVALUATION

We perform experiments to evaluate the efficacy of the proposed solution (Section 3) with refinements (Section 4) and compare it with baselines (Section 5).

6.1 Experimental Settings

We use the TPC-H benchmark with scaling factor 10. We use the following three IO-dominant queries that have been mentioned in the introduction of the paper:

(Q1) `SELECT pg_prewarm('lineitem');`
(Q2) `SELECT * FROM lineitem WHERE <p1>;`
(Q3) `SELECT * FROM lineitem WHERE <p2>.`

Although we can use more complex TPC-H queries, they are typically bound by CPU computation rather than IO latency. Given that the goal of this work is to optimize for IO performance, we choose to focus on IO-dominant queries. Table 1 summarizes the IO profiles of these queries. For each query, it reports the total number of disk blocks read, the total number of IOs, and the average number of blocks contained by an IO request. Each block is of size 8KB. Note that PostgreSQL will combine *consecutive* block reads into one single IO request, until the `io_combine_limit` (IOCL) is reached.

Hardware and Software Setups. We use a standard Azure D16s-v3 VM with 16 vCPUs and 128GB of main memory. We also use an Azure managed disk of size 1TB with the tier of "premium SSD" [53]. We run all experiments with PostgreSQL 18, which introduces support of asynchronous IO, and Ubuntu 22.04 LTS.

Configurations of PostgreSQL. We vary the following configuration knobs of PG whose settings can have an impact on the performance of the AIO subsystem, some of which have been mentioned in the introduction of this paper.

- `shared_buffers` (SB): Following common practice, we set it to 32GB, which is 25% of the total available memory of the database server used in our evaluation.
- `effective_io_concurrency` (EIC): Its default setting is 16. We change it to 64 to allow for a larger range of choosing the lookahead distance.
- `io_combine_limit` (IOCL): We use its default setting of 16, which means that each IO request can contain up to 16 contiguous disk blocks.
- `io_workers`: It specifies the number of AIO workers. We vary it in {3, 8}, where 3 is the default setting.

With these settings, the maximum distance that could be chosen is $d_{\max} = \text{EIC} \times \text{IOCL} = 64 \times 16 = 1,024$, as discussed in Section 1.2. Moreover, we enable *direct IO* by executing the SQL statement:

```
ALTER SYSTEM SET debug_io_direct TO 'data';
```

This asks the kernel to minimize caching effects for relation data.

Performance Metrics. We consider two metrics to measure the performance of the proposed solution and other baselines: (1) the "stopping distance" where an approach ends up with, and (2) the query latency with this stopping distance. Ideally, one approach is better than another if both the query latency is lower and the distance found is smaller. However, this may not always be possible. For example, one approach can find a smaller distance that results in higher query latency than the other.

Baselines. We compare our proposed approach, denoted as **Max Power**, with the baseline approaches presented in Section 5: (1) the customized version of BBR in Section 5.1 for PG AIO distance control, denoted as **BBR**; (2) the multi-armed bandit formulation in Section 5.2, denoted as **MAB-UCB**; and the Bayesian optimization formulation in Section 5.3, denoted as **GP-EI**.

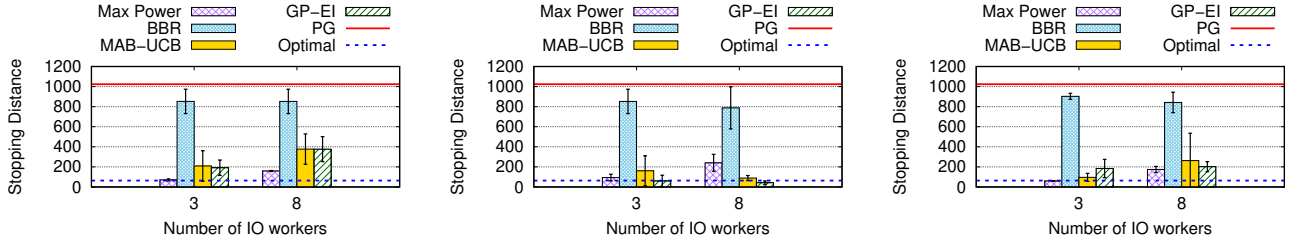
6.2 End-to-end Evaluation Results

We evaluate read streams with sequential reads and random reads. Specifically, we use query Q1 for sequential reads and use queries Q2 and Q3 for random reads (Section 6.1). For each query, we run all approaches five times and report the average distance achieved by each approach, as well as the corresponding standard deviation.

6.2.1 Sequential Reads. According to the PostgreSQL manual [60], the `pg_prewarm` module provides a convenient way to load relation data into either the operating system buffer cache or the PostgreSQL buffer cache. Therefore, Q1 results in pure sequential reads of the `lineitem` table with negligible CPU computation overhead. As a result, the query latency of Q1 is dominated by IO operations. Moreover, since the disk blocks of `lineitem` is accessed sequentially, it always allows for IO combination. That is, each IO request from Q1 contains exactly 16 disk blocks (as shown in Table 1), which is the maximum length of IO combination specified by IOCL.

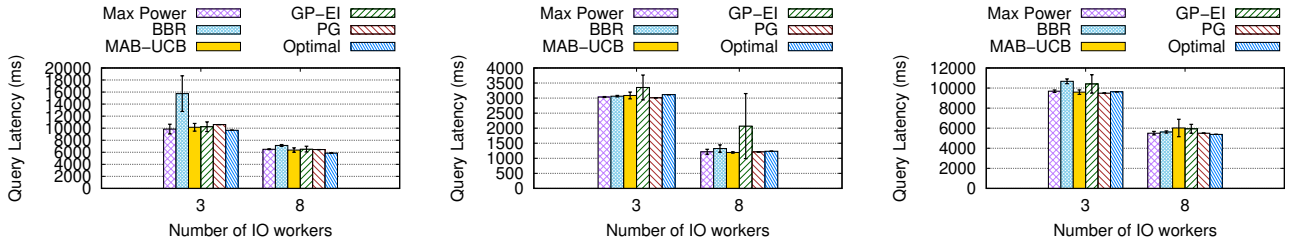
Figure 12(a) presents the (average) stopping distances of all approaches. The solid red line represents the stopping distance achieved by the current PG strategy described in Algorithm 1, which is always 1,024 based on our experimental setup. The dashed blue line represents the optimal stopping distance. We observe that **Max Power** can stop at a closer distance to the optimal one, compared to the baselines. Moreover, the standard deviation (shown as an error bar) of the stopping distance achieved by **Max Power** is also smaller than the baselines, implying a higher stability or robustness of **Max Power**. Among the baseline approaches, **BBR** reaches a large distance that is close to what PG uses. The distances reached by the two ML-based baselines, **MAB-UCB** and **GP-EI**, are smaller than the distance reached by **BBR**, but remain much larger than the distance found by **Max Power** and the optimal distance.

Figure 13(a) further presents the (average) query latency at the stopping distance for each approach. As expected, PG achieves query latency similar to that with the optimal distance, given that PG always reaches the maximum possible distance under our experimental setup. On the other hand, **Max Power** and the baselines can also achieve optimal query latency with small standard deviations (again shown as error bars), although **Max Power** stops at smaller distances compared to the baselines.



(a) `pg_prewarm` (Q1 with sequential IOs) (b) *Bitmap heap scan* (Q2, with fewer random IOs) (c) *Bitmap heap scan* (Q3, with more random IOs)

Figure 12: Stopping distance achieved by the proposed approach ("Max Power") compared to baseline approaches.



(a) `pg_prewarm` (Q1 with sequential IOs) (b) *Bitmap heap scan* (Q2, with fewer random IOs) (c) *Bitmap heap scan* (Q3, with more random IOs)

Figure 13: Query latency at the stopping distance by the proposed approach ("Max Power") compared to baseline approaches.

(a) `pg_prewarm` (Q1 with sequential IOs) (b) *Bitmap heap scan* (Q2, with fewer random IOs) (c) *Bitmap heap scan* (Q3, with more random IOs)

Figure 14: Efficacy of online sampling: each graph shows the observed throughput and corresponding confidence interval.

6.2.2 Small Number of Random Reads. Figures 12(b) and 13(b) present the stopping distance and the corresponding query latency for Q2 achieved by all approaches. Q2 is dominated by random IOs. As is evident from Table 1, the average IO size is around one disk block, indicating a limited opportunity for IO combination.

With three IO workers (i.e., the default setting of PG), both **Max Power** and **GP-EI** can find the optimal distance, although **GP-EI** exhibits a larger standard deviation, as Figure 12(b) shows. Again, **BBR** researches a large distance that is close to the maximum possible distance that PG uses. The distance found by **MAB-UCB** is much smaller than that found by **BBR** but remains larger than the optimal distance. Compared to **Max Power** and **GP-EI**, both **BBR** and **MAB-UCB** exhibit much larger standard deviations with the distances that they find.

When increasing the number of IO workers to eight, **Max Power** finds a stopping distance that is larger than the optimal distance, with a larger standard deviation compared to **MAB-UCB** and **GP-EI**. **MAB-UCB** can find the optimal stopping distance in this case, whereas **GP-EI** finds a stopping distance smaller than the optimal distance. Unfortunately, the stopping distance found by **GP-EI** results in a much higher query latency, as Figure 13(b) shows. As shown in Figure 2, using a lookahead distance that is too small can significantly increase the query latency. This suggests that one may further set a lower bound for the lookahead distance as a guardrail to prevent it from being too small. However, choosing this lower bound appropriately remains a challenge, and we leave it as one direction for future work. On the other hand, the behavior of **BBR** remains similar, which finds a distance close to the maximum possible with a large standard deviation.

6.2.3 Large Number of Random Reads. Figures 12(c) and 13(c) present the stopping distance and the corresponding query latency for Q3 achieved by all the approaches. Like Q2, Q3 is dominated by random IOs, since its average IO size is around one disk block (see Table 1). Moreover, Q3 performs more than five times random IOs compared to Q2, which also results in higher query latency.

Using three IO workers, **Max Power** achieves both optimal stopping distance and optimal query latency, with negligible standard deviations. **MAB-UCB** finds a slightly larger stopping distance with slightly larger standard deviation. **GP-EI** finds a much larger stopping distance with a much larger standard deviation. The corresponding query latency of **GP-EI** is also higher than the optimal query latency, with a larger standard deviation, as Figure 13(c) shows. Once again, **BBR** yields a stopping distance that is close to the maximum possible, also with elevated query latency.

When increasing the number of IO workers to eight, both **Max Power** and **GP-EI** achieve similar stopping distances that are larger than the optimal one, with similar standard deviations. Their corresponding query latency and its standard deviation are also similar,

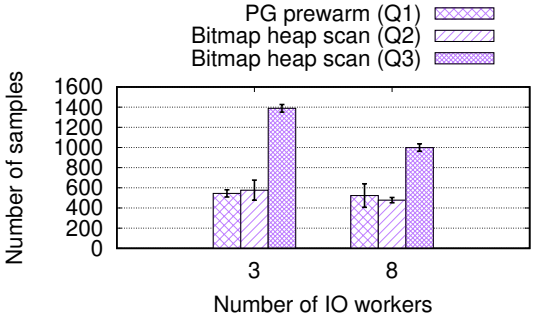


Figure 15: Number of IO waits in online sampling.

as Figure 13(c) shows. Compared to **Max Power** and **GP-EI**, **MAB-UCB** finds a larger stopping distance with a much larger standard deviation, indicating some instability in its behavior. **BBR** again finds a stopping distance close to the maximum distance allowed. Finally, both **Max Power** and the baselines achieve the optimal query latency in this case.

6.3 Efficacy of Online Sampling

Each graph in Figure 14 presents the mean observed throughput with its confidence interval as the lookahead distance increases, using the online sampling mechanism presented in Section 4.1. Here we set $\alpha = 0.05$, which implies a confidence level of $1-\alpha = 0.95$. We observe a tiny confidence interval that is almost invisible, indicating that the mean observed throughput is very close to the ground-truth mean throughput when online sampling ends.

Figure 15 further presents the average number of IO waits, as well as its standard deviation (shown as an error bar), taken in the online sampling process, for each of the three queries Q1, Q2, and Q3. We observe that the number of IO waits taken by Q1 and Q2 is similar, whereas Q3 requires more IO waits. Nevertheless, the number of IO waits required for all three queries is negligible compared to the total number of IOs (see Table 1).

7 DISCUSSION AND FUTURE WORK

Multiple Read Streams. We have been focusing on the problem of tuning the lookahead distance of a single read stream. In practice, it is common to see multiple read streams running concurrently within a PG instance. An interesting question is whether the approach proposed in this paper remains effective in such multi-stream setups. We have tested some simple setups, such as running two concurrent `pg_prewarm` queries over different TPC-H tables, and we do not see the difference in terms of the chosen stopping distance by our approach. However, a more systematic study is desirable, especially in a production PG database deployment with real-world customer workloads, which remains a challenge on Azure Database for PostgreSQL due to reasons such as data compliance and privacy. We therefore leave this for future work.

The Case for Using `io_uring`. We have focused on the default `worker` option offered by the PG AIO subsystem. However, our proposed approach can also be applied to the `io_uring` option, because our approach is designed on top of the read stream abstraction and thus is oblivious to the underlying implementation of the PG AIO subsystem. `io_uring` is a high-performance Linux asynchronous IO framework that drastically improves efficiency by reducing system calls and context switches through shared, lock-free ring buffers between user space and the kernel. As a result, PG AIO performance (e.g., throughput) could be further improved for Linux kernels that support `io_uring`. Nevertheless, as reported by recent work [33], careful tuning is required to achieve a performance improvement of 14% when PG is integrated with `io_uring`. Therefore, although we would expect a larger optimal lookahead distance to match the improved IO throughput with `io_uring`, a thorough investigation is needed to better understand the impact of the lookahead distance, which we leave for future work.

8 RELATED WORK

Prefetching. Database systems use a *buffer pool* to cache data pages that are frequently accessed. As a result, *prefetching* pages from the storage system that will be accessed in the near future into the buffer pool can have significant benefit by reducing the number of "page faults" that can significantly slow down query processing. There has been extensive work on improving prefetching for database and storage systems [4, 12, 14–16, 20, 22, 25, 30, 35, 45–47, 52, 65, 67, 70, 72, 74, 75, 82, 83, 86]. One key challenge in prefetching is to speculate or predict the next pages to be fetched from the storage system, based on historical access patterns. As a result, it is typically viewed as an online sequential prediction problem, and solutions to this problem include data compression [20, 45, 75], analytic cost modeling [30, 67, 86], access pattern matching [22, 46, 47, 70, 72, 82], and recently machine learning [4, 12, 15, 16, 25, 52, 83]. As we noted earlier in the paper, the problem of tuning lookahead distance of the PG AIO subsystem is different from this line of work on prefetching, as the pages to be fetched are known and do not require speculation or prediction. The new challenge is to *minimize* the time required to fetch these pages with minimum buffer pool occupancy.

Buffer Pool Management. In addition to prefetching, another related problem in buffer pool management is *cache replacement*. Specifically, when a page is brought in (either on demand or via prefetching), if there is no empty frame or slot left in the buffer pool, then the database system needs to select a *victim* page and evict it to make room for the new page. The question is to design a good replacement policy to minimize the impact of evicting the old page, and there have been many proposals [1, 18, 34, 56, 57]. Although we do not consider the impact on existing buffer pool pages when bringing more pages into memory with a larger lookahead distance, we recognize this as an interesting direction for future work. For example, existing work on the cost modeling of cache replacement policies [55, 80] may be useful. Moreover, one may want to further consider the gap between the time when a page is brought in and the time when the page is *used* by the caller of the read stream.

Flow Control in Computer Networking. The study of flow control in computer networking can date back several decades. An interesting line of work in this area is the study of the power function [7, 32, 37, 39–41], which inspired our work in this paper. As was pointed out by Kleinrock [41], the problem of maximizing the power of a system coincides with the problem of finding its optimal operational point, which lays the theoretical foundation for the development of BBR [9, 10, 23, 31, 61, 63, 69], one of the state-of-the-art TCP congestion control protocols designed for Internet-scale network communication. From a control theory point of view, BBR is also an instance of *max-plus control* [6, 29, 64]. As a result, the

principles of max-plus control can be applied to further improve the BBR-style baseline studied in this paper for tuning lookahead distance (Section 5.1). In addition, there are other TCP congestion control protocols, such as additive increase and multiplicative decrease (AIMD) [17], CUBIC [28], Copa [5], and Timely [54]. While BBR requires monitoring both throughput and latency to compute the "bandwidth-delay product (BDP)," some of these protocols, such as Copa and Timely, are delay-based and only require monitoring latency. It is possible that similar delay-based mechanisms can be developed for tuning the lookahead distance of the PG AIO subsystem. We leave this as a fertile ground for future exploration.

Database Knob Tuning. In recent years, there has been intensive research on the problem of database knob tuning (see [85] for a survey), where the goal is to find the values for the database system configuration parameters (a.k.a. *knobs* in the literature), such as the `shared_buffers` for PG, that can optimize the performance of a target workload. Existing solutions are based on machine learning (ML) technologies such as Bayesian optimization [3, 11, 24, 36], reinforcement learning [2, 49, 76, 84], and most recently large language models [26, 48, 71]. Moreover, some recent approaches [26, 76] further include *index tuning* as part of the database knob tuning problem, although index tuning is generally viewed as a different problem that itself has been studied extensively [8, 13, 21, 43, 44, 62, 66, 73, 77, 78, 81]. Our work on tuning the lookahead distance of the PG AIO subsystem is a special instance of the general database knob tuning problem, although the lookahead distance is currently not exposed as a PG knob that can be tuned externally. Our proposed approach is analytic and outperforms ML-based approaches, as is evident in our evaluation. This provides an alternative thinking to the general problem of database knob tuning. Although ML technologies are prevalent in existing solutions, simpler and more analytic approaches may remain competitive or even more effective for tuning certain types of knobs. This is therefore a potential new direction for future research.

9 CONCLUSION

We presented a new approach to tuning the lookahead distance for asynchronous IOs in PostgreSQL. It is inspired by the well established theory of optimal flow control in computer networking. We developed a surrogate of the power function of a (queuing) system, the maximization of which gives the optimal operational point. We further used online sampling to reduce the uncertainty in the observed throughput and thus the value of the surrogate power function. Experimental evaluations demonstrate the efficacy of our proposed approach for IO-dominant queries with both sequential and random reads, which outperforms the current strategy used by PostgreSQL as well as other baseline approaches based on BBR, multi-armed bandits, or Bayesian optimization.

ACKNOWLEDGMENTS

We thank our colleagues from the Microsoft PostgreSQL team, Daniel Gustafsson, Melanie Plageman, Krishnakumar Ravi, and Nazir Bilal Yavuz, for their help in the early stages of this work.

A PROOFS

A.1 Proof of Theorem 1

We have the derivative of $P(\gamma)$ as $P'(\gamma) = \frac{1}{T} - \frac{\gamma \cdot T'}{T^2}$. It follows that

$$\begin{aligned} P''(\gamma) &= \left(\frac{1}{T}\right)' - \left(\frac{\gamma \cdot T'}{T^2}\right)' \\ &= -\frac{1}{T^2} \cdot T' - \frac{(\gamma \cdot T')' \cdot T^2 - (\gamma \cdot T') \cdot (T^2)'}{T^4} \\ &= -\frac{T'}{T^2} - \frac{T' + \gamma \cdot T''}{T^2} + \frac{\gamma \cdot 2T \cdot (T')^2}{T^4} \\ &= \frac{T \cdot (-\gamma \cdot T'' - 2T') + 2\gamma \cdot (T')^2}{T^3}. \end{aligned}$$

Substituting $\gamma^* = \frac{T}{T'}$, we obtain

$$\begin{aligned} P''(\gamma^*) &= \frac{T \cdot (-\frac{T}{T'} \cdot T'' - 2T') + 2 \cdot \frac{T}{T'} \cdot (T')^2}{T^3} \\ &= \frac{-T^2 \cdot \frac{T''}{T'} - 2T \cdot T' + 2T \cdot T'}{T^3} \\ &= -\frac{T''}{T \cdot T'} = -\frac{h''(\gamma^*)}{h(\gamma^*) \cdot h'(\gamma^*)}. \end{aligned}$$

Clearly, $h(\gamma^*) > 0$ since delay/latency must be positive, $h'(\gamma^*) > 0$ since h is increasing, and $h''(\gamma^*) > 0$ since h is strongly convex. As a result, we have $P''(\gamma^*) < 0$ and therefore $P(\gamma)$ achieves its maximum at γ^*. In other words, the stationary point with throughput γ^* satisfying $\gamma^* = \frac{h(\gamma^*)}{h'(\gamma^*)}$ yields the *maximum power*.

A.2 Proof of Theorem 2

We have the derivative of the surrogate $P_s(\lambda)$ as

$$P_s'(\lambda) = \frac{\gamma' \cdot \lambda - \gamma}{\lambda^2} = \frac{\gamma'}{\lambda} - \frac{\gamma}{\lambda^2}.$$

Setting $P'(\lambda) = 0$ gives

$$\gamma' \cdot \lambda = \gamma.$$

Therefore, the stationary point λ^* satisfies

$$\lambda^* = \frac{\gamma}{\gamma'} = \frac{g(\lambda^*)}{g'(\lambda^*)}.$$

Next, consider the second derivative $P_s''(\lambda)$ at λ^*. We have

$$\begin{aligned} P_s''(\lambda) &= \left(\frac{\gamma'}{\lambda}\right)' - \left(\frac{\gamma}{\lambda^2}\right)' \\ &= \frac{\gamma'' \cdot \lambda - \gamma'}{\lambda^2} - \frac{\gamma' \cdot \lambda^2 - (\lambda^2)' \cdot \gamma}{\lambda^4} \\ &= \frac{\gamma''}{\lambda} - \frac{\gamma'}{\lambda^2} - \frac{\gamma'}{\lambda^2} + \frac{2\lambda \cdot \gamma}{\lambda^4} \\ &= \frac{\gamma''}{\lambda} - \frac{2\gamma'}{\lambda^2} + \frac{2\gamma}{\lambda^3}. \end{aligned}$$

Substituting $\lambda^* = \frac{\gamma}{\gamma'}$, we obtain

$$\begin{aligned} P_s''(\lambda^*) &= \frac{\gamma'' \cdot \gamma'}{\gamma} - \frac{2\gamma'^3}{\gamma^2} + \frac{2\gamma \cdot \gamma'^3}{\gamma^3} \\ &= \frac{\gamma'' \cdot \gamma'}{\gamma} - \frac{2\gamma'^3}{\gamma^2} + \frac{2\gamma'^3}{\gamma^2} \\ &= \frac{\gamma'' \cdot \gamma'}{\gamma} = \frac{g''(\lambda^*) \cdot g'(\lambda^*)}{g(\lambda^*)}. \end{aligned}$$

Clearly, $g(\lambda^*) > 0$ since throughput is positive, $g'(\lambda^*) > 0$ since g is increasing, and $g''(\lambda^*) < 0$ since g is strongly concave. As a result, $P_s''(\lambda^*) < 0$, and therefore, $P_s(\lambda)$ reaches its maximum at the lookahead distance λ^*.

REFERENCES

[1] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. 1971. Principles of Optimal Page Replacement. *J. ACM* 18, 1 (1971), 80–93.

[2] Dana Van Aken et al. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (2021), 1241–1253.

[3] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*. ACM, 1009–1024.

[4] Ibrahim Umit Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. 2023. Improving Storage Systems Using Machine Learning. *ACM Trans. Storage* 19, 1 (2023), 9:1–9:30.

[5] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *NSDI*. 329–342.

[6] François Baccelli and Dohy Hong. 2000. TCP is max-plus linear: and what tells us on its throughput. In *SIGCOMM*. 219–230.

[7] Kadaba Bharath-Kumar and Jeffrey M. Jaffe. 1981. A new approach to performance-oriented flow control. *IEEE Trans. Commun.* 29, 4 (1981), 427–435.

[8] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD*. 227–238.

[9] Yi Cao, Arpit Jain, Kriti Sharma, Aruna Balasubramanian, and Anshul Gandhi. 2019. When to use and when not to use BBR: An empirical analysis and evaluation study. In *ACM IMC*. 130–136.

[10] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, 5 (2016), 20–53.

[11] Stefano Cereda et al. 2021. CGPTuner: a Contextual Gaussian Process Bandit Approach for the Automatic Tuning of IT Configurations Under Varying Workload Conditions. *Proc. VLDB Endow.* 14, 8 (2021), 1401–1413.

[12] Chandranil Chakraborttii and Heiner Litz. 2020. Learning I/O Access Patterns to Improve Prefetching in SSDs. In *ECML PKDD*. 427–443.

[13] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.

[14] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving hash join performance through prefetching. *ACM Trans. Database Syst.* 32, 3 (2007), 17.

[15] Yu Chen, Yong Zhang, Jiacheng Wu, Jin Wang, and Chunxiao Xing. 2021. Revisiting Data Prefetching for Database Systems with Machine Learning Techniques. In *ICDE*. 2165–2170.

[16] Giovanni Cherubini, Yusik Kim, Mark A. Lantz, and Vinodh Venkatesan. 2017. Data Prefetching for Large Tiered Storage Systems. ICDM, 823–828.

[17] Dah-Ming Chiu and Raj Jain. 1989. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems* 17, 1 (1989), 1–14.

[18] Hong-Tai Chou and David J. DeWitt. 1986. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica* 1, 3 (1986), 311–336.

[19] Mark Claypool. 2017. BBR Implementation 2. https://github.com/mark-claypool/bbr.

[20] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. 1993. Practical Prefetching via Data Compression. In *SIGMOD*. 257–266.

[21] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *Proc. VLDB Endow.* 4, 6 (2011), 362–372.

[22] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. 2007. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *USENIX ATC*. 261–274.

[23] Rebecca Drucker, Gauri Baraskar, Aruna Balasubramanian, and Anshul Gandhi. 2024. BBR vs. BBRv2: A Performance Evaluation. In *IEEE COMSNETS*. 379–387.

[24] Songyun Duan et al. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (2009), 1246–1257.

[25] Gaddisa Olani Ganfure, Chun-Feng Wu, Yuan-Hao Chang, and Wei-Kuan Shih. 2020. DeepPrefetcher: A Deep Learning Framework for Data Prefetching in Flash Storage Devices. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39, 11 (2020), 3311–3322.

[26] Victor Giannakouris and Immanuel Trummer. 2025. λ-Tune: Harnessing Large Language Models for Automated Database System Tuning. *Proc. ACM Manag. Data* 3, 1 (2025), 2:1–2:26.

[27] Google. 2016. BBR Implementation. https://github.com/google/bbr.

[28] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Oper. Syst. Rev.* 42, 5 (2008), 64–74.

[29] Bernd Heidergott, Geert Jan Olsder, and Jacob W van der Woude. 2006. *Max Plus at work: modeling and analysis of synchronized systems: a course on Max-Plus algebra and its applications*. Vol. 13. Princeton University Press.

[30] Herodotos Herodotou and Elena Kakoulli. 2023. Cost-based Data Prefetching and Scheduling in Big Data Platforms over Tiered Storage Systems. *ACM Trans. Database Syst.* 48, 4 (2023), 11:1–11:40.

[31] Mario Hock, Roland Bless, and Martina Zitterbart. 2017. Experimental evaluation of BBR congestion control. In *IEEE ICNP*. 1–10.

[32] Jeffrey M. Jaffe. 1981. Flow control power is non-decentralizable. *Perform. Evaluation* 1, 1 (1981), 95.

[33] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, Viktor Leis, and Carsten Binnig. 2025. High-Performance DBMSs with io_uring: When and How to use it. *CoRR* abs/2512.04859 (2025).

[34] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*. 439–450.

[35] Mahesh Kallahalla and Peter J. Varman. 2001. Optimal prefetching and caching for parallel I/O systems. In *SPAA*. 219–228.

[36] Konstantinos Kanellis et al. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. *Proc. VLDB Endow.* 15, 11 (2022), 2953–2965.

[37] Parviz Kermani and Leonard Kleinrock. 1980. Dynamic Flow Control in Store-and-Forward Computer Networks. *IEEE Trans. Commun.* 28, 2 (1980), 263–271.

[38] Leonard Kleinrock. 1975. *Queuing systems, volume i: Theory*. John Wiley & Sons New York.

[39] Leonard Kleinrock. 1978. On flow control in computer networks. In *Proceedings of the International Conference on Communications*, Vol. 2.

[40] Leonard Kleinrock. 1979. Power and deterministic rules of thumb for probabilistic problems in computer communications. In *International Conference on Communications*, Vol. 3. 43–1.

[41] Leonard Kleinrock. 2018. Internet congestion control using the power metric: *Keep the pipe just full, but no fuller*. *Ad Hoc Networks* 80 (2018), 142–157.

[42] Leonard Kleirock. 1976. *Queuing systems volume 2: Computer applications*. Chichester, USA: Wiley.

[43] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.

[44] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. 2022. SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning. In *EDBT*. 2:155–2:168.

[45] P. Krishnan and Jeffrey Scott Vitter. 1998. Optimal Prediction for Prefetching in the Worst Case. *SIAM J. Comput.* 27, 6 (1998), 1617–1636.

[46] Tom M. Kroeger and Darrell D. E. Long. 2001. Design and Implementation of a Predictive File Prefetching Algorithm. In *USENIX ATC*. 105–118.

[47] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. 2016. Lynx: a learning linux prefetching mechanism for SSD performance model. In *NVMSA*. 1–6.

[48] Jiale Lao et al. 2025. GPTuner: An LLM-Based Database Tuning System. *SIGMOD Rec.* 54, 1 (2025), 101–110.

[49] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130.

[50] Yang Li, Yu Shen, Wentao Zhang, Yuanwei Chen, Huaijun Jiang, Mingchao Liu, Jiawei Jiang, Jinyang Gao, Wentao Wu, Zhi Yang, Ce Zhang, and Bin Cui. 2021. OpenBox: A Generalized Black-box Optimization Service. In *KDD*. 3209–3219.

[51] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *JMLR* 23 (2022).

[52] Edson Ramiro Lucas Filho, Lun Yang, Kebo Fu, and Herodotos Herodotou. 2023. Streaming machine learning for supporting data prefetching in modern data storage systems. In *Proceedings of the First Workshop on AI for Systems*. 7–12.

[53] Microsoft. 2025. Azure managed disk types. https://learn.microsoft.com/en-us/azure/virtual-machines/disks-types.

[54] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*. 537–550.

[55] Victor F. Nicola, Asit Dan, and Daniel M. Dias. 1992. Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing. In *SIGMETRICS*. ACM, 35–46.

[56] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *SIGMOD*. 297–306.

[57] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1999. An Optimality Proof of the LRU-*K* Page Replacement Algorithm. *J. ACM* 46, 1 (1999), 92–112.

[58] PostgreSQL. 2025. Asynchronous I/O (AIO). https://www.postgresql.org/about/featurematrix/detail/asynchronous-io-aio/.

[59] PostgreSQL. 2025. IO Methods. https://www.postgresql.org/docs/18/runtime-config-resource.html.

[60] PostgreSQL. 2025. Preload relation data into buffer cache. https://www.postgresql.org/docs/current/pgprewarm.html.

[61] Simon Scherrer, Markus Legner, Adrian Perrig, and Stefan Schmid. 2022. Model-based insights on the performance, fairness, and stability of BBR. In *ACM IMC*. 519–537.

[62] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies. In *ICDE*. 1238–1249.

[63] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle. 2018. Towards a Deeper Understanding of TCP BBR Congestion Control. In *IFIP Networking Conference and Workshops*. 109–117.

[64] Bart De Schutter and Ton J. J. van den Boom. 2001. Model predictive control for max-plus-linear discrete event systems. *Autom.* 37, 7 (2001), 1049–1056.

[65] Elizabeth AM Shriver, Christopher Small, and Keith A Smith. 1999. Why does file system prefetching work?. In *USENIX ATC*. 71–84.

[66] Tarique Siddiqui and Wentao Wu. 2023. ML-Powered Index Tuning: An Overview of Recent Progress and Open Challenges. *SIGMOD Rec.* 52, 4 (2023), 19–30.

[67] Alan Jay Smith. 1978. Sequentiality and Prefetching in Database Systems. *ACM Trans. Database Syst.* 3, 3 (1978), 223–247.

[68] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS*.

[69] Linchuan Tang. 2024. BBR Fairness Evaluation Using NS-3. *arXiv preprint arXiv:2410.22560* (2024).

[70] Nancy Tran and Daniel A. Reed. 2004. Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching. *IEEE Trans. Parallel Distributed Syst.* 15, 4 (2004), 362–377.

[71] Immanuel Trummer. 2024. DB-BERT: making database tuning tools "read" the manual. *VLDB J.* 33, 4 (2024), 1085–1104.

[72] Ahsen J. Uppal, Ron Chi-Lung Chiang, and H. Howie Huang. 2012. Flashy prefetching for high-performance flash drives. In *IEEE MSST*. 1–12.

[73] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*. 101–110.

[74] Peter J. Varman and Rakesh M. Verma. 1999. Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems. *IEEE Trans. Parallel Distributed Syst.* 10, 12 (1999), 1262–1275.

[75] Jeffrey Scott Vitter and P. Krishnan. 1996. Optimal Prefetching via Data Compression. *J. ACM* 43, 5 (1996), 771–793.

[76] Junxiong Wang et al. 2021. UDO: Universal Database Optimization using Reinforcement Learning. *Proc. VLDB Endow.* 14, 13 (2021), 3402–3414.

[77] Xiaoying Wang, Wentao Wu, Vivek R. Narasayya, and Surajit Chaudhuri. 2025. Esc: An Early-Stopping Checker for Budget-aware Index Tuning. *Proc. VLDB Endow.* 18, 5 (2025), 1278–1290.

[78] Xiaoying Wang, Wentao Wu, Chi Wang, Vivek R. Narasayya, and Surajit Chaudhuri. 2024. Wii: Dynamic Budget Reallocation In Index Tuning. *Proc. ACM Manag. Data* 2, 3 (2024), 182.

[79] Wikipedia. 2025. Linux IO Uring. https://en.wikipedia.org/wiki/Io_uring.

[80] Wentao Wu, Yun Chi, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *Proc. VLDB Endow.* 6, 10 (2013), 925–936.

[81] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek R. Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. 2022. Budget-aware Index Tuning with Reinforcement Learning. In *SIGMOD*. 1528–1541.

[82] Xiaofei Xu, Zhigang Cai, Jianwei Liao, and Yutaka Ishikawa. 2020. Frequent Access Pattern-based Prefetching Inside of Solid-State Drives. In *DATE*. 720–725.

[83] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A Learned Prefetcher for Cache Invalidation in LSM-tree based Storage Engines. *Proc. VLDB Endow.* 13, 11 (2020), 1976–1989.

[84] Ji Zhang et al. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD*. 415–432.

[85] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2023. Automatic Database Knob Tuning: A Survey. *IEEE Trans. Knowl. Data Eng.* 35, 12 (2023), 12470–12490.

[86] Shengan Zheng, Hong Mei, Linpeng Huang, Yanyan Shen, and Yanmin Zhu. 2017. Adaptive Prefetching for Accelerating Read and Write in NVM-Based File Systems. In *ICCD*. 49–56.