Concurrency Control for Conflicting Experiments

Daniel Ting
Meta

Houssam Nassif
Meta

1 Introduction

Large-scale experimentation platforms run thousands of experiments concurrently. Some of these experiments modify shared components or parameters and cannot be run together. For example, a UI element's text and background should not be set to the same color, and a user cannot receive two different search rankings [7]. Experimentation platforms often accommodate these conflicting experiments by introducing a notion of "layers" [1], which ensure experiments in the same layer receive mutually exclusive treatment populations. Putting conflicting treatments in the same layer ensures that no conflicting treatments can be administered.

However, some experiments modify multiple shared components or parameters. Workarounds, such as creating uber-layers that merge multiple layers, introduce bottlenecks by forcing more experiments to queue up in the same uber-layer. Some of these experiments may target different layers so their treatments can be applied simultaneously, but the uber-layer treats them as conflicting regardless. These additional conflicts can make it impossible to run experiments at the desired scale.

To address this, we draw inspiration from computer systems and recast the problem of running many experiments as a concurrency-control problem. By using locking mechanisms on shared resources, we both ensure experiments cannot deliver conflicting treatments and ensure that only genuinely conflicting experiments are blocked from running together.

2 Primitives

In concurrency control, conflicts arise from contention on shared resources. Layers [1] can be seen as defining one type of shared resource; however, under a layer scheme, an experiment can only use a single shared resource type. An experiment that needs a sample size of n must acquire n experimental units, for example users, where it can apply the desired treatments. Thus, the actual shared resources are (experimental unit, resource type) pairs. To prevent conflicting treatments from being applied to an experimental unit, each shared resource has a *mutex*: a lock that may be held by at most one treatment cohort at a time, granting that cohort exclusive rights to set the corresponding parameter for the units it holds, analogous to locks in classical database concurrency control [3, 4].

Our main idea is to allow experiments to use multiple types of shared resources and to lock only the actual resources used by the experiment. More formally, an experiment $\mathcal{E}$ defines its resource requirements as

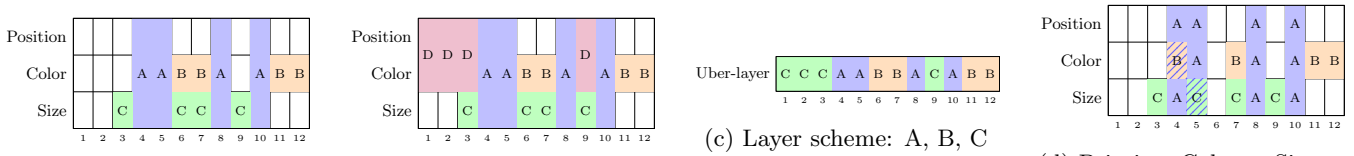
- the number of units n needed, and
- a set of resource requirements $\mathcal{R}$.

Allocation consists of three stages:

- **Population identification**: finding all units for which none of the required resources are locked and no other conflicts can be introduced;
- **Randomization**: sampling from the population and randomly assigning units to treatment cohorts;
- **Lock acquisition**: locking the mutexes for the (sampled unit, resource type) pairs actually used by each treatment cohort.

Releasing a treatment cohort unlocks all of its mutexes.

For example, consider a UI element with three resource types: Position, Color, and Size, and a population divided into 12 units. Experiment A changes all three; experiment B changes only Color; experiment C changes only Size. Later a fourth experiment D wishes to change Position and Color. Each experiment requests one-third of the units. Figure 1 illustrates how concurrency control allows more experiments to be run and allocates experiment D while a Layer-only schema cannot.



(a) Concurrency control (b) D added: Position, Color (c) Layer scheme: A, B, C (d) Priority: Color > Size > Position

Figure 1: Example table of shared resources. Rows are resource types; columns are 12 units, each labeled with the experiment currently holding it. (a) B and C can share units leaving 2 units completely free. (b) Concurrency control has enough room to allocate experiment D. (c) Under a layer scheme, A's use of all three resources merges Position, Color, and Size into one uber-layer, The 3 experiments use up all 12 units, leaving none for D. (d) Under a priority scheme, A's Color and Size are silently changed by other experiments on units 4 and 5 (striped cells). It delivers a UI element that is different from any intended design because the treatments should not be modified independently of each other.

Population identification and lock acquisition are performed as a single atomic allocation step, analogous to validating a snapshot immediately before committing writes, so that no other experiment can acquire a conflicting lock between the two stages. Because every lock an experiment will ever need is acquired at once during this step rather than incrementally over the experiment's lifetime, the scheme avoids the deadlocks that can otherwise arise from incremental multi-resource locking [3].

For any experiment that is a randomized controlled trial, randomization already involves identifying the eligible population and sampling from it. The contribution of concurrency-control ideas here is a ledger that makes it easy to track which part of the population is eligible for inclusion in a new experiment.

Defining resources. In many experimentation systems, the treatment for a cohort is determined by a set of configurable parameters. These parameters naturally define resource types, since a parameter cannot be set to two different values at once for the same unit. Additional resources may be defined beyond the parameters themselves, for example to represent shared UI real estate or downstream system capacity.

Resources should also not be too cumbersome to track. Since the number of shared resources can grow as $O(n_u \cdot n_r)$, where n_u is the number of units and n_r is the number of resource types, it is beneficial to keep n_u small. A standard method is to use buckets, turning fine-grained units into coarse units: rather than tracking individual users, units are divided into n_B buckets via a random hash. This parameter controls a tradeoff between system concerns, which prefer small n_B, and statistical concerns, which prefer large n_B.

Statistical considerations from coarsening. Coarsening affects the ability of an experiment to eliminate imbalance in potential confounders between test and control groups. When a potential confounder is independent of the bucketing, the hash used for bucketing still induces a valid randomization over the fine-grained units, so coarsening introduces no bias in this case. It does, however, inflate variance: because treatment status is identical for every unit in a bucket, correlation among a bucket's outcomes—whether from correlated baseline behavior (e.g., a shared device or social connection) or from heterogeneous treatment response—no longer averages out the way it would under independent, unit-level randomization. This is the classical cluster-randomization *design effect*, DEFF $\approx 1+(\bar{m}-1)\rho$, where $\bar{m}$ is the average bucket size and ρ is the intraclass correlation of the outcome within a bucket [5]. A well-mixed hash keeps ρ small for any single experiment; the more realistic bias risk instead comes from *reusing* the same bucket assignment across many sequential experiments, which can let a bucket's treatment history become a confounder in its own right.

3 Alternatives

Layers or exclusion groups. The standard method for handling conflicting treatments is to use layers [1], where each layer consists of a collection of experiments assigned disjoint populations. These are also called exclusion groups, universes, or namespaces [2]. Other platforms documented as supporting layers include Optimizely [12], Statsig [13], GrowthBook [14], LaunchDarkly [15], and Eppo [16]. Because a layer is a single resource type, an experiment touching several resources forces every layer it touches to be merged into one uber-layer. In the running example, A touches Position, Color, and Size, so those three layers must merge into a single uber-layer that B and C also fall into, since they touch Color and Size respectively. A, B, and C already fill this uber-layer with disjoint one-third slices (Figure 1c), so no more than three experiments can run regardless of how few resources each one touches; unlike in Figure 1b, there is no room left for D.

Multiplicative exclusion groups. Several platforms, including AB Tasty [18] and Amplitude [17], support assigning an experiment to multiple exclusion groups. Each exclusion group acts as an independent targeting condition: if experiment A is in group X with 50% allocation and in group Y with 40% allocation, then A receives $0.5 \times 0.4 = 20\%$ of the units. Traffic share therefore decreases dramatically as an experiment is included in more exclusion groups.

Treatment priorities. Due to the large number of potentially conflicting experiments, an experimentation platform may implement a priority system for its layers. In addition to layers, the system would have a hard-coded set of resource types. When conflicting treatments target the same resource type, the treatment belonging to the highest-priority layer is used; this resolution happens independently for each resource type, so the original combination of treatments assigned to a unit may be partially modified [8, 9, 10]. Under this scheme, lower-priority layers may not actually receive their intended treatment, diluting the estimated treatment effect. This dilution can occur even when using triggering, also known as exposure logging [6], since higher-priority experiments can still take units from lower-priority experiments and effectively end and dilute treatment for those previously exposed units. The scheme resolves hard conflicts among treatments, but does so by modifying the treatment an experiment specified. This can still be suitable behavior for some experiments but should not be applied to experiments if individual treatment components cannot be modified independently from others. This could result in unintended treatments as illustrated in Figure 1d. AB Tasty similarly implements a prioritization mechanism in addition to its exclusion groups [19].

Real-time allocation. Netflix's experimentation platform dynamically allocates a user to an experiment only if the user is not already in a conflicting experiment [11]; details of how experiments are ordered for this check are not public. While this approach does not silently modify an experiment's treatment, it adds latency to treatment delivery, and estimating the sample size for an experiment becomes more difficult.

References

[1] D. Tang, A. Agarwal, D. O'Brien, and M. Meyer. Overlapping experiment infrastructure: More, better, faster experimentation. In *KDD '10*, Washington, DC, USA, 2010.

[2] E. Bakshy, D. Eckles, and M. S. Bernstein. Designing and deploying online field experiments. In *WWW '14*, 2014.

[3] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, 1976.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[5] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne. Controlled experiments on the web: survey and practical guide. *Data Mining and Knowledge Discovery*, 18(1):140–181, 2009.

[6] R. Kohavi, A. Deng, B. Frasca, R. Longbotham, T. Walker, and Y. Xu. Trustworthy online controlled experiments: Five puzzling outcomes explained. In *KDD '12*, 2012.

[7] F. Kong, Y. Li, H. Nassif, T. Fiez, S. Chakrabarti, and R. Henao. Neural insights for digital marketing content design. In *KDD'23*, pages 4320–4332, 2023.

[8] M. Malmir, M. A. Radwan, H. Nassif, and M. A. Bayir. Beyond IPS: Reliable Counterfactual Evaluation in Multi-Stage Ad Systems without Logged Propensities. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2026.

[9] M. Malmir, H. Nassif, D. N. Shaikh, T. Rahgooy, and M. A. Bayir. Impression Share Prediction: An Offline Evaluation Task for Ranking Systems. In *Conference on Recommender Systems (RecSys)*, 2026.

[10] M. A. Radwan, Q. Lanners, J. Zhang, S. Karakulak, H. Nassif, and M. A. Bayir. Counterfactual evaluation of ads ranking models through domain adaptation. In *Workshops of Conference on Recommender Systems (RecSys)*, 2024.

[11] Netflix Technology Blog. It's all a/bout testing: The Netflix experimentation platform. `https://netflixtechblog.com/its-all-a-bout-testing-the-netflix-experimentation-platform-4e1ca458c15`, 2016. Accessed 2026-08-26.

[12] Optimizely. Use mutual exclusion. `https://docs.developers.optimizely.com/feature-experimentation/docs/use-mutual-exclusion`. Updated 2026-07-23. Accessed 2026-08-26.

[13] Statsig. Layers overview. `https://docs.statsig.com/experiments/layers-overview`. Accessed 2026-08-26.

[14] GrowthBook. Mutual exclusion and namespaces. `https://docs.growthbook.io/lib/js#mutual-exclusion`. Accessed 2026-08-26.

[15] LaunchDarkly. Creating mutually exclusive experiments. `https://launchdarkly.com/docs/guides/experimentation/mutually-exclusive`. Accessed 2026-08-26.

[16] Eppo. Mutual exclusion. `https://docs.geteppo.com/feature-flagging/concepts/mutual_exclusion/`. Accessed 2026-08-26.

[17] Amplitude. Mutually exclusive experiments. `https://amplitude.com/docs/feature-experiment/advanced-techniques/mutually-exclusive-experiments`. Accessed 2026-08-26.

[18] AB Tasty. Mutually exclusive experiments feature. `https://docs.abtasty.com/web-experimentation-and-personalization/campaign-creation-and-dashboard/mutually-exclusive-experiments-feature`. Accessed 2026-08-26.

[19] AB Tasty. How to use campaign prioritization. `https://docs.abtasty.com/web-experimentation-and-personalization/campaign-creation-and-dashboard/how-to-use-campaign-prioritization`. Accessed 2026-08-26.